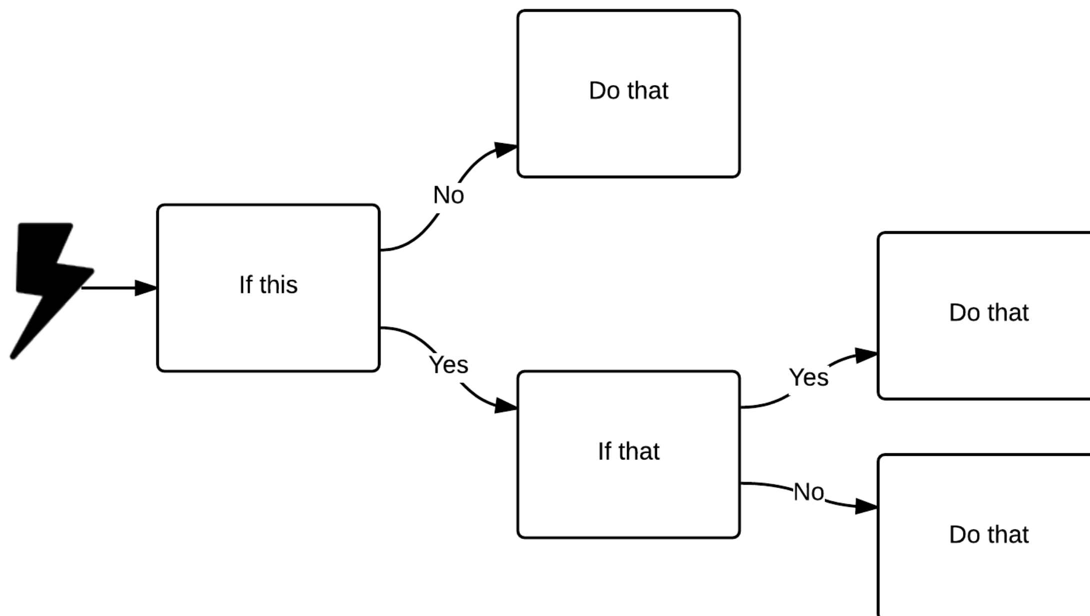
CHALMERS
UNIVERSITY OF TECHNOLOGY



Utveckling av en Dynamisk Händelsemotor

Design och Implementering av en händelsemotor i C++

Examensarbete inom Högskoleingenjörsprogrammet i Datateknik

Tobias Edvardsson, Marcus Flyckt

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA UNIVERSITET
Göteborg, Sverige 2016

Utveckling av en Dynamisk Händelsemotor

Design och Implementering av en händelsemotor i C++

Tobias Edvardsson, Marcus Flyckt

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Universitet
Göteborg, Sverige 2016

Utveckling av en Dynamisk Händelsemotor
Design och Implementering av en händelsemotor i C++
Tobias Edvardsson, Marcus Flyckt

© Tobias Edvardsson, Marcus Flyckt, 2016.

Handledare: Carolin Hellestam, ICOMERA, Lennart Hansson, CHALMERS
Examinator: Peter Lundin, Data- och informationsteknik

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Universitet
412 96 Göteborg
Telefon +46 31 772 1000

Förord

Rapporten är en del av det examensarbete vi genomfört under våren 2016 för högskoleingenjörsprogrammet för Datateknik vid Chalmers Tekniska Högskola, Göteborg. Examensarbetet har utförts hos uppdragsgivaren Icomera.

Handledare hos uppdragsgivaren har varit Carolin Hellestam. Handledare på Chalmers Tekniska Högskola har varit Lennart Hansson.

Vi vill tacka Lennart för de tips och diskussioner som hjälpt arbetet framåt under projektets gång. Även stort tack Icomera, framförallt Carolin och Roger, för deras visade stöd under utvecklingen. Med allas hjälp har vi tagit fram en produkt som vi är nöjda med och fått ihop en rapport vi kan känna oss stolta över.

Tobias Edvardsson och Marcus Flyckt, Göteborg, juni 2016

Sammanfattning

När ett företag växer och fler och fler system skapas och integreras i dess systemmiljö uppkommer det ett behov av att centralisera och standardisera hanteringen av olika händelser kopplade till företaget eller dess system. Denna rapport har undersökt det använda och beprövade fenomenet 'Business Rules Engine' och därifrån tagit inspiration för att utveckla en ny regelstyrd händelsemotor. Motorn som tagits fram är enkel att använda, utforma och anpassa efter företagets eller andra specifika behov. Den erbjuder ett enkelt sätt att koppla händelser till beteenden. Beteendena definieras i valfritt programmeringsspråk och kan styras med villkorssatser, även dem skrivna i godtyckligt programmeringsspråk. Det har även genomförts en undersökning kring de problem som finns i att exekvera externa kommandon från C++-kod, och tagits fram förslag på hur dessa kan motarbetas. För att underlätta framtagningen av en prototyp har även ett grafiskt webbaserat gränssnitt utvecklats. Detta för att underlätta hanteringen av den data som bearbetas av händelsemotorn.

Nyckelord: händelsemotor, skriptexekvering, regelstyrd, dynamisk, centraliserad, business rules.

Abstract

When a company grows and more and more systems are created and integrated into its system environment, a need to centralize and standardize the management of various events that are of relevance to the company or or its systems arises. This report studies the well-tried phenomenon of 'Business Rules Engines' and uses those ideas to develop and build a new rule based event engine. The engine developed is easy to use, design and customize for a company's, or other, specific needs. It provides an easy way to connect events to behaviors. The behaviors can be defined in any programming language and can be controlled with conditional statements, also written in any programming language. A study was also performed regarding issues while executing external commands from a C++ code base and proposes how they can be mitigated. To ease the prototype development a web-based interface was developed. This was done in order to ease the handling of data to be processed by the event engine.

Nyckelord: händelsemotor, skriptexekvering, regelstyrd, dynamisk, centraliserad, business rules.

Ordlista

- **AJAX** - 'Asynchronous JavaScript and XML'. Teknik för att hämta och skicka data dynamiskt via JavaScript.
- **Action** - Exekverbar åtgärd som användare själva kan skriva i valfritt skriptspråk.
- **API** - 'Application Programming Interface'. Specifikation hur man kan använda ett objekt.
- **Back-end** - Den del av processen där logiken arbetar.
- **Check** - Exekverbar kontroll som användare själva kan skriva i valfritt skriptspråk.
- **Demon** - En Demon är i ett operativsystem en speciell form av process som inte har någon dedikerad input- eller output-kanal.
- **Event** - En händelse som aktiverar ett flöde av checkar och actions.
- **Front-end** - Del av system som användaren ser och använder för att manipulera processen. Grafiska gränssnitt, websidor eller liknande.
- **Interface** - Specifikation av ett objekts API.
- **Interpreter** - En interpreter är ett program som får programkod som input och sedan genomför de instruktioner som står beskrivna i koden.
- **Kernel** - Kärnan i ett operativsystem. Lagret mellan applikation och hårdvara.
- **ORM** - 'Object-relational mapping'. Koppla databas tabeller till objekt inom programmering för att underlätta användandet av databaser inom koden man skriver.
- **Overhead** - Overhead är all den tid och resurser som går åt vid en process som inte direkt är till nytta för processen.

Innehåll

1 Introduktion **1**
1.1 Bakgrund 1
1.2 Syfte 1
1.3 Mål 1
1.4 Avgränsning 2

2 Teknisk Bakgrund **3**
2.1 Business Rules och Business Rules Engine 3
2.1.1 Business Rule (BR) 3
2.1.2 Business Rules Engine (BRE) 3
2.2 Inter-Process Communication (IPC) 3
2.2.1 UNIX-socket 4
2.2.2 Message Queue (MQ) 4
2.3 Event-Driven Architecture (EDA) 4
2.3.1 Delar inom EDA 5
2.4 Systemanrop 5
2.5 Databaser 5
2.5.1 Structured Query Language - SQL 5
2.5.2 SQLite 5
2.6 Webutveckling 6
2.6.1 Flask 6
2.6.2 Peewee 6
2.7 Google Test 6

3 Utvecklingsmetod **7**
3.1 Projekt 7
3.1.1 Agilt utvecklande 7
3.1.2 Parprogrammering 7
3.2 Förarbete 7
3.2.1 Förstudie 7
3.2.2 Undersökning av dynamiska skriptexekveringsmetoder 8
3.2.3 Framtagning av prototyp 8

4 Design och Framtagning av en Händelsemotor **9**
4.1 Förstudieresultat 9
4.2 Skriptexekveringsundersökning 9

4.3 Back-end 12
4.3.1 Framtagning av prototyp 12
4.3.2 Låg sammankoppling 12
4.3.3 Judge 12
4.3.4 Prosecutor 13
4.3.5 Controller 13
4.3.6 Simpel/Komplex 14
4.3.7 Engine 15
4.3.8 Första iterationen 16
4.3.9 Andra iterationen 17
4.3.10 Tredje och slutgiltiga iterationen 18
4.3.11 Jämförelse med klassisk EDA och BRE 19
4.3.12 Skillnader mot klassisk BRE 19
4.4 Front-end 19
4.4.1 Användarvänlighet 19
4.4.2 Grafiskt web-gränssnit 20
4.4.3 HTTP-API 21
4.4.4 Databas 22
4.5 Tester 23
4.5.1 Automatiserade tester 23
4.5.2 Funktionella tester 23
4.6 Integrering med befintliga system 23

5 Resultat **25**
5.1 Förundersökning 25
5.2 Utveckling 25

6 Slutsats **27**
6.1 Användarvänlighet 27
6.2 Fortsatt utveckling 27
6.2.1 Front-end 27
6.2.2 Back-end 27
6.3 Slutdiskussion 28
6.3.1 Hållbar Utveckling 28

Litteraturförteckning **29**

1
Introduktion

Uppsatsen handlar om framtagning av en dynamisk händelsemotor. I introduktionen kommer bakgrund, syfte och mål beskrivas.

1.1 Bakgrund

I en verksamhet där den huvudsakliga produkten är programutveckling medföljer ofta att många processer som utvecklas blir beroende av varandra, har behovet att nyttja funktionalitet från olika källor eller vid fel genomföra vissa åtgärder på ett automatiserat tillvägagångssätt.

Genom att ta fram en produkt som vid en händelse kan utföra en kedja av kontroller och beteenden kan man uppnå ny funktionalitet utan att behöva skriva om befintliga programvaror. Produkten ska också bidraga till en tydligare överblick då idag utspridd funktionalitet centraliseras.

På Icomera där exjobbet genomfördes fanns initialt tanken att projektet skulle ha fokus kring övervakning och automatiserad felsökning. Icomera är i behov av ett system som implementerar detta eller en mer generell lösning på ett centraliserat händelsehanteringssystem. Detta kan bidra till automatisering av vissa processer som idag utförs manuellt.

Även möjlighet att standardisera och samla de skript som används inom olika delar i verksamheten var önskvärt.

1.2 Syfte

Syftet med projektet är att bygga ett verktyg som kan underlätta för Icomeras verksamhet, i synnerhet deras supportavdelning. Även att möjliggöra strukturering och standardisering av befintliga utspridda skript som används för dagligt underhåll inom företagets verksamhet.

1.3 Mål

Projektet strävar efter att uppnå följande punkter:

- En händelsemotor som innefattar olika händelser med regelkedjor
- Undersöka befintliga lösningar av liknande motorer
- Ta fram ett effektivt regelsyntax
- Direktaktiverade och schemalagda händelser

- Databas med händelser och regler, samt de skript som behövs
- Central version, anpassad för verksamheten
- Ordentligt testad backend
- Webgränssnitt för enkel hantering av databasinformation

1.4 Avgränsning

Då mycket av slutprodukten kommer bero på att man utvecklar ett effektivt och bra GUI skulle detta kunna vara ett projekt i sig. Projektet har avgränsats till att enbart innefatta framtagning av själva händelsemotorn och utveckla stöd så att ett väl genomarbetat GUI kan utarbetas och appliceras senare. Ett enklare GUI skall dock implementeras för att enklare kunna utföra funktionella tester och hantera den data som händelsemotorn bearbetar. Händelsemotorn skall även kunna rapportera händelser vidare till tredje part via ett framtida API.

2
Teknisk Bakgrund

I detta kapitel finns de tekniska kunskaper som behövs för att förstå rapportens resterande innehåll.

2.1 Business Rules och Business Rules Engine

2.1.1 Business Rule (BR)

En Business Rule, hädanefter kallad regel, är

> "[...]ett uttryck som definierar eller åtstramar någon aspekt av ett företag. Den är till för att försäkra företagsstruktur eller för att kontrollera eller påverka beteendet av företaget. En regel är atomisk, det vill säga, den kan inte brytas ner ytterligare"[5].

Omformulerat, och mer anpassat mot ett datasystem, så är en regel en datastruktur innehållandes information om systemets nuvarande status.

2.1.2 Business Rules Engine (BRE)

Business Rules Engines, hädanefter kallad regelmotorer, är

> "[...] mjukvaruapplikationer som innehåller definitioner av regler"[3]

Även detta är en väldigt vid definition som behöver brytas ner ytterligare. Denna rapport kommer definiera en regelmotor som 'En applikation som enligt givna regler genererar beteenden som uppfyller dessa regler'.

2.2 Inter-Process Communication (IPC)

IPC refererar till olika mekanismer för att skicka information mellan olika processer som körs på något datorsystem. Posix definierar flera typer av IPC.

Signaler: Används för att signalera att en händelse har inträffat.

Pipes (Rör): Kopplar utdata från en process till indata för en annan process.

Socket: Används för att överföra data mellan processer. Detta kan ske både internt i en dator eller distribuerat över ett nätverk.

Fillåsning: Möjliggör för processer att "låsa" regioner av en fil för att hindra andra processer från att läsa eller modifiera dessa områden.

Meddelandeköer: Kompletta meddelanden med information skickas mellan processer. Grundtanken är att producenter producerar information och konsu-

menter konsumerar information. Denna metod kan leda till en särkopplad arkitektur där producenter och konsumenter inte behöver känna till varandra.

Semaforer: Grundläggande synkronisering mellan processer. Denna metod är mer begränsad då man endast har tillgång till numeriska, eller vid binära semaforer booleska, värden.

Delat minne: En process skapar trådar som kommunicerar via gemensamma variabler. Denna metod kräver mest utav utvecklaren. Semaforer behöver oftast använda för att undvika problem då flera trådar försöker läsa eller skriva till samma minnesposition.[10, 6]

2.2.1 UNIX-socket

En socket är en abstraktion för kommunikation mellan processer. Kommunikationen kan ske både över ett nätverk men också internt i en maskin. En Unix-socket är ett exempel på en socket som hanterar kommunikation internt i en dator. Socketen utnyttjar UNIX filsystem för att skicka meddelanden mellan processer. Dataöverföring till en socket kan ske både med TCP-strömmar och datagram. [2].

2.2.2 Message Queue (MQ)

En message queue, eller meddelandekö, kan ses som en länkad lista av meddelanden. Trådar eller processer kan lägga meddelanden i kön eller hämta meddelanden därifrån. Meddelanden som skrivs till kön ges också en viss prioritet. En fördel med meddelandeköer över socketar är att en socket är beroende av att en server, eller konsument, aktivt väntar på meddelanden. Meddelandeköer däremot existerar oberoende av producenter och konsumenter.[1]

2.3 Event-Driven Architecture (EDA)

EDA är tätt sammankopplat med 'service-oriented architecture' (SOA). SOA är ett koncept för att särskilja en viss tjänst, eller ett visst arbete, från den faktiska entiteten som utför arbetet. Klienten som begär en tjänst känner bara till vad tjänsten är och hur man begär den. Den enda som känner till hur tjänsten är implementerad är tjänsten själv. Tjänster aktiveras vanligtvis direkt eller indirekt av mänskliga användare av ett system, men de kan också anropas från en av systemets genererade händelser. En händelse är

> [...]någonting noterbart som sker inom eller utanför ett system. En händelse kan signalera ett innestående eller antågande problem, en signifikant möjlighet, en tröskelnivå som nåtts eller någon annan avvikelse.[8]

En händelse består av två delar enligt EDA. En rubrik och en kropp. Rubriken innehåller metainformation om händelsen, som namn, typ, tidpunkt och vem som skapade händelsen. I kroppen finns information om vad som hänt. I ett EDA-system leds en genererad händelse automatiskt till intresserade parter. Dessa parter kan vara både mänskliga och autonoma program eller skript. Denna arkitektur leder till ett väldigt frikopplat system. Den som skapar händelsen vet inget mer om systemet

förutom att händelsen skapats. EDA kan delas in i fyra separerade delar. Händelsegeneratorer, händelsekanaler, händelsebearbetning och händelseaktivitet.

2.3.1 Delar inom EDA

Händelsegeneratorer är de entiteter som genererar och rapporterar händelse till systemet. De kan bestå av sensorer, sändare eller program i en dator. Transporten mellan händelsegeneratorer och händelsebearbetning går genom en händelsekanal. Händelsekanalen består ofta av något meddelandesystem. I händelsebearbetningen jämförs händelsen mot systemets uppsatta regler och systemet får reda på vilka parter som är intresserade av denna händelse.

En händelsemotor kan vara komplex eller simpel. I en simpel händelsemotor evalueras varje enskild händelse individuellt, samtidigt som en komplex motor tar hänsyn till tidigare händelser. Händelseaktivitet är de tjänster eller aktiviteter som kallas eller initieras som följd av händelsen. Det kan vara allt från skickande av mail till avancerad programlogik.[8]

2.4 Systemanrop

Ett systemanrop är en metod för att, på ett kontrollerat sätt, tillåta en process att anropa en tjänst i det underliggande systemet, eller Kerneln.

Exempel på systemanrop är då man från en process vill skapa en ny process, behöver exekvera ett annat program eller skript, när man behöver utföra I/O-hantering eller vid IPC. Då en CPU ofta delar upp datorns minne i en applikationsdel och en kernel-del, måste anrop mellan system och applikation flytta en mängd data, parametrar och returvärden, mellan system- och applikationsminne. Detta är en process som ofta kräver betydligt mer tid och resurser än ett vanligt funktionsanrop inom samma minnesrymd.[6]

2.5 Databaser

2.5.1 Structured Query Language - SQL

Structured Query Language, hädanefter kallat SQL, är ett programmeringsspråk skrivet för att hantera relationsdatabaser. SQL härstammar ifrån relationsalgebran och består av av olika operationer för att hämta, skapa, uppdatera eller ta bort information.

2.5.2 SQLite

SQLite är en databasmotor för en relationsdatabas som jobbar mot en fil i det lokala filsystemet. Till skillnad från vanliga databasmotorer kräver SQLite ingen serverprocess utan arbetar direkt mot lokal disk. SQLite är en väldigt liten databasmotor och populär för inbyggnad i program för att därmed tillåta manipulation av filer med

hjälp av SQL-kommandon. Då SQLite ej kräver någon installation eller serverprocess är det ett bra verktyg vid utveckling av programvaror som har användning av någon form av relationsdatabas.[12]

2.6 Webutveckling

2.6.1 Flask

Flask är ett ramverk för webutveckling byggt i Python.

> ”Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions. And before you ask: It’s BSD licensed!”[9]

2.6.2 Peewee

Peewee är en ORM byggt för Python. Det är en simpel variant av en ORM som gör det enkelt att ta fram och börja använda på kort tid. Peewee stöder flera databaser bland annat SQLite som är använt i detta projekt.

> ”Peewee is a simple and small ORM. It has few (but expressive) concepts, making it easy to learn and intuitive to use.”[7]

2.7 Google Test

Google Test är Googles egna ramverk för testning av C++-kod. Den är uppdelad i två olika ramverk, Google Test och Google Mock. Google Test innehåller grundläggande funktionalitet för assertioner och fixturer, medan Google Mock tillåter så kallad *mockning* av objekt, det vill säga skapandet av objekt som för omvärlden ser ut som riktiga objekt men i verkligheten saknar funktionalitet.

3 Utvecklingsmetod

3.1 Projekt

3.1.1 Agilt utvecklande

Projektet genomfördes agilt. Att arbeta agilt innebär att man under projektets gång kontinuerligt har en diskussion om vad för olika funktioner och implementationsdetaljer det skall läggas fokus på.

Till en början sätts en avgränsning på vad projektet skall uppnå och därefter prioriteras de funktioner som skall implementeras allt eftersom projektbilden växer fram. Med detta medföljer att projektet blir förfinat under hela projekttiden och genom detta även slutprodukten.

3.1.2 Parprogrammering

Under projektet genomfördes utvecklingen till stor del genom parprogrammering. Genom att i par skriva gemensam kod anses det i allmänhet att resultatet blir både bättre och nås snabbare än de fall man sitter och programmerar enskilt. Parprogrammering sker genom att man turas om att skriva kod samtidigt som den andra observerar vad som skrivs och kommer med förslag och åsikter kring koden. Genom att ha en kontinuerlig kommunikation under programmeringstillfället delar programmerarna både nya kunskaper och hjälper varandra utvecklas inom det som skrivs. Det finns uppskattningar att koden som skrivs tar ca 15% längre tid men innehåller 15% mindre fel. [11]

3.2 Förarbete

3.2.1 Förstudie

För att applikationen skulle få en väl genomtänkt struktur och att den ej skulle göra något som redan fanns inleddes projektet med en förstudie. I förstudien lades mycket tid på att hitta befintliga teorier kring liknande lösningar och system som påminner om det projektet ska uppnå.

Informationsinhämtningen gick till genom att litterära källor söktes i tillgängliga databaser och på annat håll.

3.2.2 Undersökning av dynamiska skriptexekveringsmetoder

Då systemet i hög grad bygger på dynamisk exekvering av externa skript, gjordes en enkel undersökning över vilka exekveringsmetoder som gav bäst resultat med avseende på tidsåtgång för en mängd exekveringar. Undersökningen fokuserade på tre olika metoder för att exekvera skript skrivna i Python från en C++-miljö.

3.2.3 Framtagning av prototyp

Projektet har som förklarat i avsnitt 3.1 utförts agilt, detta har medfört flera iterationer av programvaran justerade efter diskussioner och möten med handledarna.

Iterationer

För att få en dynamik i både projekt och slutlig produkt har det genomförda arbetet flera gånger diskuterats och anpassats efter handledarnas önskemål. Efter genomförda studier och möten har nya användningsområden och målsättningar utvecklats och presenterats i olika iterationer av systemet för att slutligen nå en nivå som uppfyller de krav och önskemål framtagna för projektet.

Gränssnitt

För att kunna påvisa användningsområden har utvecklingen av ett grafiskt gränssnitt varit nödvändigt. Detta för att enklare visualisera hur och vad man kan genomföra med hjälp utav Händelsemotorn men även för att underlätta hantering av den data Händelsemotorn bearbetar i sitt flöde.

4

Design och Framtagning av en Händelsemotor

Projektets slutliga form är något som via ett agilt projektgenomförande bidragit till en väl anpassad lösning. Mycket av det som utvecklats under projektet har gjorts genom att man använt parprogrammering som metod i framtagning av programvaran.

I det här kapitlet presenteras undersökningen rörande skriptexekvering samt projektets implementations- och designdetaljer.

4.1 Förstudieresultat

Projektet inleddes med en förstudie där 'Business rules' 2.1 var något av de första som denna bidrog med och något som projektet i flera steg nyttjat för att få till en bra slutprodukt. I sökandet av andra liknande produkter hittades flera system som byggde på idén med 'Business rules' men inget som uppfyllde de krav projektet hade utan jobbade snarare mot en statisk databas för att utföra sina beslut.

Förstudien ledde också till fördjupade kunskap inom C++-utveckling i en Linuxmiljö.

4.2 Skriptexekveringsundersökning

I undersökningen skulle varje metod anropa ett testskript ett ökande antal gånger i följd. Testskriptet såg ut enligt följande.

```
# script.py
def run():
        print "Ran"

if __name__ == '__main__':
        run();
```

Ren Python

Ren Python betyder i detta sammanhang att skriptet skrivet i Python importeras och exekveras av ett annat Python-skript. Detta skulle i praktiken innebära att

systemet som ska exekvera skripten först måste generera ett *motorskript* som sedan tar över ansvaret för exekveringen.

```
# python_call_test.py
import script
import sys
for i in range(int(sys.argv[1])):
        script.run()
```

Python.h

Då Python finns implementerat i C finns det möjlighet att inkludera hela språket i egen C- eller C++-kod. Detta innebär att systemet får en egen tolk, eller Interpreter, kapabel att exekvera Python-strängar. I förlängningen innebär detta att man aldrig behöver lämna sin egen applikation för att be operativsystemet kalla sin version av Python-Interpretern.

```
// python_h_call_test.c
#include <Python.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]){
        int scriptfd;
        int i;
        char code[1024];
        scriptfd = open("script.py", O_RDONLY);
        read(scriptfd, code, 1024);
        Py_SetProgramName(argv[0]);
        Py_Initialize();
        for (i = 0; i < atoi(argv[1]); ++i)
        {
                if(PyRun_SimpleString(code) != 0){
                        printf("Could not run script\n");
                }
        }
        return 0;
}
```

system()

Användandet av systemanropet system() innebär att man ger operativsystemet en kommandorad att evaluera och exekvera. Detta leder till att exekveringen av det egna programmet tillfälligt lämnar den egna koden och låter operativsystemet exe-

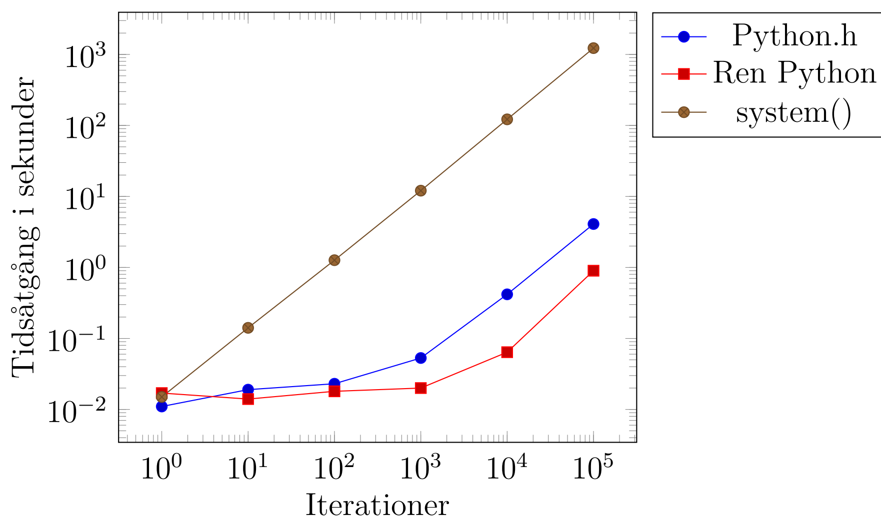
kvera skriptet. I händelsemotorns fall innebär detta att den i sitt systemanrop ber operativsystemet starta systemets Python-Interpreter tillsammans med den skriptfil som ska exekveras.

```
// system_call_test.c
#include <stdio.h>

int main(int argc, char *argv[]){
        int i;
        for (i = 0; i < atoi(argv[1]); ++i)
        {
                system("python script.py");
        }
}
```

Undersökning

Undersökningen gick till på så sätt att varje metod fick exekvera testskriptet en, tio, hundra, tusen, tiotusen och hundratusen gånger. Tiden som mättes var den verkliga tidsåtgången. Alltså inte CPU-tid för just den processen. Resultatet är utskrivet i grafen nedan. Grafen är ritad i logaritmisk skala i båda axlar.



Resultat

Det framgår tydligt att användandet av systemanrop bör undvikas. Den stora skillnaden mellan system() och de två övriga kan förklaras med att vid användandet av systemanrop startas en ny Python-tolk upp för varje iteration. Detta skapar Overhead som tillsammans med den extra tid systemanropet medför ger en betydligt snabbare ökning i exekveringstid.

4.3 Back-end

4.3.1 Framtagning av prototyp

Utvecklingen inleddes i samband med förstudien i avsnitt 4.1, för att försöka få en bättre förståelse kring det valda programmeringsspråket C++. Idéer testades successivt via parprogrammering och modeller togs fram för att kunna genomföra det som efterfrågats. Genom att skissera fram olika flödesdiagram på hur processen kunde fungera togs slutligen en modell som uppfyllde de krav och diskussioner som diskuterats under projektets möten fram.

4.3.2 Låg sammankoppling

En viktig punkt att utgå ifrån då programmet designats är att processerna skall vara så löst sammankopplade som möjligt. De skall göra sin uppgift och sedan ej bry sig om eller vara beroende på andra processer. Händelsemotorn försöker följa denna princip i stor grad då verktyget är tänkt att kunna användas inuti flera olika system. Händelsemotorn kommer ej vara beroende av andra processer än de som processen i sig bygger på.

4.3.3 Judge

Algoritmen för regelevaluering ligger i Judge-klassen. Algoritmen, beskriven i pseudokod nedan, går till enligt följande:

1. Sortera alla regler och lägg dem i en lista
2. Hämta nästa regel från listan
3. Finns det en Check så exekvera den
4. Var checken inte falsk så exekvera Action.
5. Nästa steg är ett *flödesbeslut* som avgörs av reglernas flödes-parametrar. Det sätts totalt tre parametrar per regel. En avgör vad som händer då Checken är falsk, och de resterande två bestämmer vad som händer då Action är sann respektive falsk. En parameter kan vara någon av
    - Fortsätt till nästa regel (gå till steg 2)
    - Hoppa till godtycklig regel (gå till steg 3 och använd denna nya regel)
    - Hoppa ur algoritmen (händelsen slutar evalueras)

Här är algoritmen i pseudokod:

```
while det finns fler regler do
    hämta regel;
    if regel.check är sann then
        if regel.action är sann then
            flödesbeslut enligt regel.actionTrue;
        else
            flödesbeslut enligt regel.actionFalse;
        end
    else
        flödesbeslut enligt regel.checkFalse;
    end
end
```

Att evaluera en Check eller Action innebär exekvering av det associerade skriptet och tolkning av skriptets returvärde som sant eller falskt.

4.3.4 Prosecutor

Regelhanteringen skiljer sig åt mellan vanlig händelsehantering med Judge-objekt och schemalagda händelser som hanteras av Prosecutor-objekt. Den mest markanta skillnaden är att Prosecutor-objekt endast hanterar Checkar. Algoritmen, beskriven i pseudokod nedan, går till enligt följande:

1. Sortera alla regler och lägg dem i en lista
2. Hämta nästa regel från listan
3. Var checken sann så gå till steg 2. Aktivera annars händelsen.

Här är algoritmen i pseudokod:

```
while det finns fler regler do
    hämta regel;
    if regel.check är sann then
        fortsätt
    else
        hoppa ur och aktivera händelse
    end
end
```

4.3.5 Controller

Controller-objektet är en central punkt som innehåller funktionalitet för att ta emot och expediera meddelanden. Det är detta objekt som ansvarar för att hålla reda på systemets tillstånd. Här tas besluten om när motorn ska startas om eller avslutas.

Nedan följer en sammanfattning av Controller-objektets expedieringsfunktion

uttryckt i pseudokod.

```
while för alltid do
    ta emot meddelande;
    if meddelande är systemmedelande then
        avsluta eller starta om enligt meddelande
    else
        if meddelande innehåller schemalagt event then
            expediera till rätt Prosecutor-objekt
        else
            if meddelande innehåller vanligt event then
                expediera till rätt Judge-objekt
            else
                okänt meddelande, ignorera
            end
        end
    end
end
```

4.3.6 Simpel/Komplex

Den fram tills nu beskrivna motorn är fortfarande en simpel händelsemotor. Det vill säga, den tar inte hänsyn till tidigare händelser då den hanterar nya händelser. För att motorn ska anses vara komplex behöver den logik för att evaluera en händelses aktuella tillstånd.

Projektets system löser detta genom att lägga ansvaret för att evaluera en händelses tillstånd på varje händelses associerade Judge-objekt. Detta kräver ännu ett steg i regelhanteringen i ett Judge-objekt. Ett Judge-objekt måste, innan den går in i regel-evaluerings-loopen, först avgöra om det alls är aktuellt för händelsen att aktiveras.

En nackdel med att lägga hela ansvaret på enskilda Judge-objekt är att händelseminnet återställs om ett objekt behöver laddas om. Återställning av Judge-objekt sker då skript eller regler blivit uppdaterade utifrån systemet och ett systemmeddelande med 'starta om'-kommando tagits emot av Controller-objektet. Får att åtgärda detta lagras minnet av en händelse istället i Controller-objektet. Det är fortfarande upp till varje Judge-objekt att avgöra om reglerna ska evalueras eller ej, men först måste det objektet gå till Controller-objektet och hämta senaste minnet av händelsen.

Flapping/Waiting

Systemet tillåter två typer av komplex händelsehantering, Flapping och Waiting. Flapping-händelser evalueras endast om händelsens medskickade parameter har ändrats sen dess förra aktivering. Waiting-händelser evalueras endast om en viss tid passerat sedan händelsens förra aktivering.

4.3.7 Engine

Den modul som i vår design kallas 'Engine' innehåller den statiska kod och logik som tillåter de ovan beskrivna objekten att kommunicera och interagera med varandra på ett korrekt sätt. Här finns till exempel all logik för vad en initiering eller omstart av motorn innebär, eller hur information hämtas från en extern databas in till systemet. Här finns även kopplingarna mellan händelser och Judge- eller Prosecutor-objekt beskrivna. När ett Prosecutor- eller Controller-objektet behöver aktivera ett Judge-objekt för evaluering av en händelse går detta via Engine-modulen.

Loggning

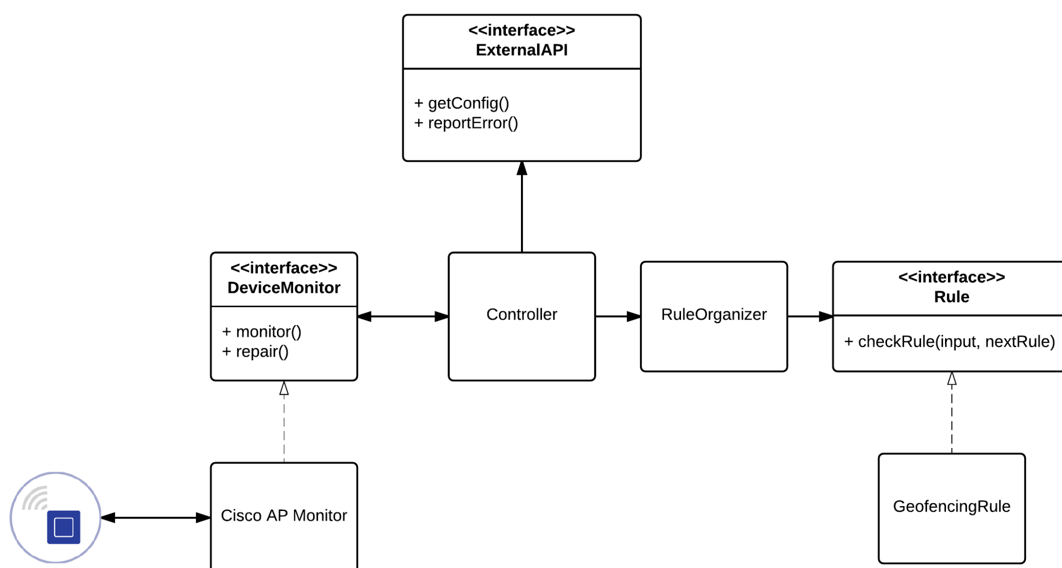
Under den tidiga perioden av systemets utveckling kördes programmet rakt i en egen konsol. Detta medförde att utskrifter från systemet, och de skript systemet anropar, skrivs ut i den konsolen. Denna utskrift kan sedan användas för felsökning och vidareutveckling. När systemet senare började köras som en Demon försvårades denna form av felsökning. Det blev då nödvändigt att införa loggning. All loggning i systemet sköts via Engine-modulen.

Loggningen i systemet baseras på händelsenamn. Varje händelse får en egen textfil där det går att följa vägen från att den registreras av Controller-objektet till att händelsens associerade skript exekveras. Även utskrifterna från skripten loggas ihop med händelsen för att underlätta felsökning av dessa.

4.3.8 Första iterationen

I första iterationen låg fokus på att alla processer som genererar händelser skulle vara tätt knutna till ett DeviceMonitor-objekt, se figur 4.1 för ett övergripande diagram. Genom att ett DeviceMonitor-objekt skulle vara i tät kommunikation med det centrala Controller-objektet kunde man då via ett knutet RuleOrganizer-objekt och en kedja med regler utföra kontroller, åtgärder och skicka rapporter via dessa objekt. I den första iterationen låg fokus kring att skapa olika Interface för både DeviceMonitor-objekt och Rules.

I denna iteration fanns även en idé om att bygga regelverket som en pipeline. Den grundläggande tanken var att pipelinen skulle initieras av ett RuleOrganizer-objekt men sedan helt sköta sig själv. Varje anrop på en regel skulle innehålla en referens till nästkommande regel. En regel kunde då välja att, när den var färdig, antingen bryta kedjan eller kalla på nästa regel. Om även sista regeln i kedjan valde att kalla på nästa regel så skulle anropet gå tillbaka till RuleOrganizer-objektet som sedan skulle rapportera felet vidare till ett externt system.

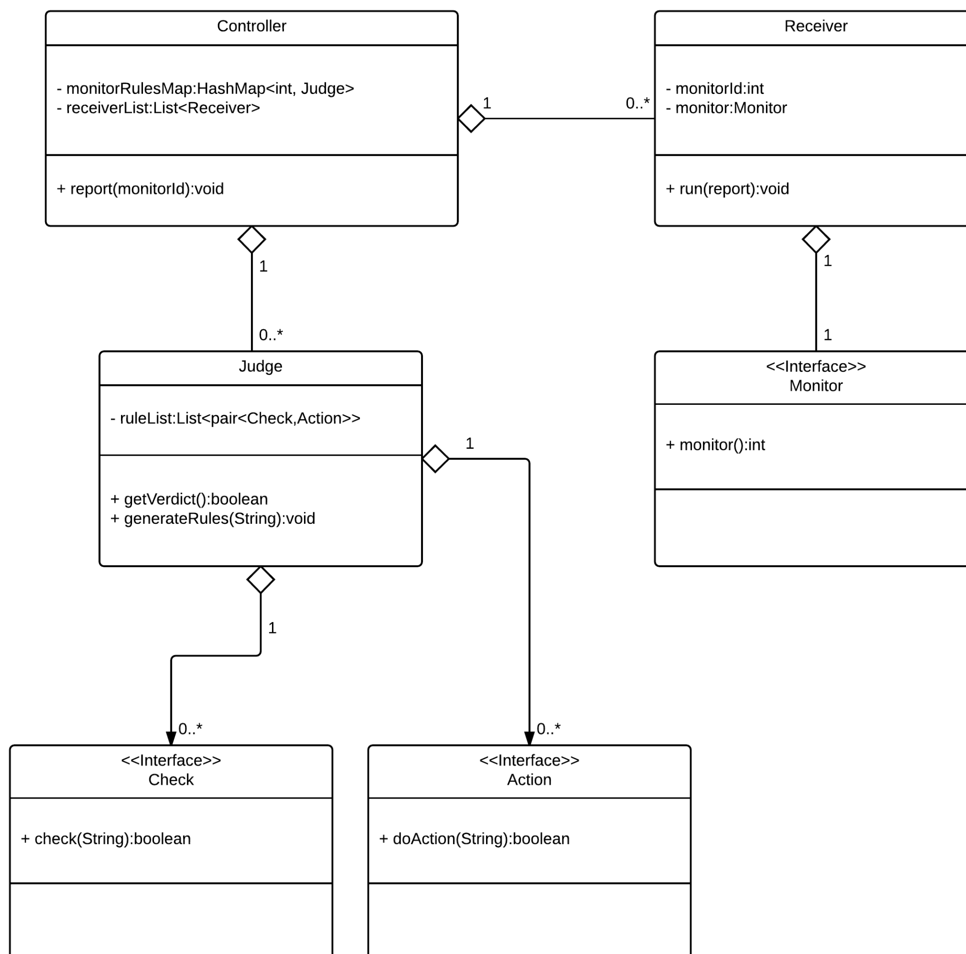


Figur 4.1: Diagram iteration 1

4.3.9 Andra iterationen

Vid andra iterationen infördes idén att ha händelser kopplade till ett Judge-objekt. En händelse kopplas direkt mot ett Judge-objekt vilket utför de kontroller och åtgärder relaterade till den händelsen. Den andra iterationen behöll fortfarande tanken kring att ha tätt knutna monitorer och via dessa aktivera de händelser associerade till ett Judge-objekt. Se figur 4.2.

I detta skede implementerades även kommunikation via en UNIX-socket vilket möjliggjorde kommunikation mellan ett Receiver-objekt och ett Controller-objekt. Genom att använda en UNIX-socket kunde man nu även nyttja Controller-objektet externt. Detta möjliggjorde använding av Händelsemotorn för tredje parts programvaror.

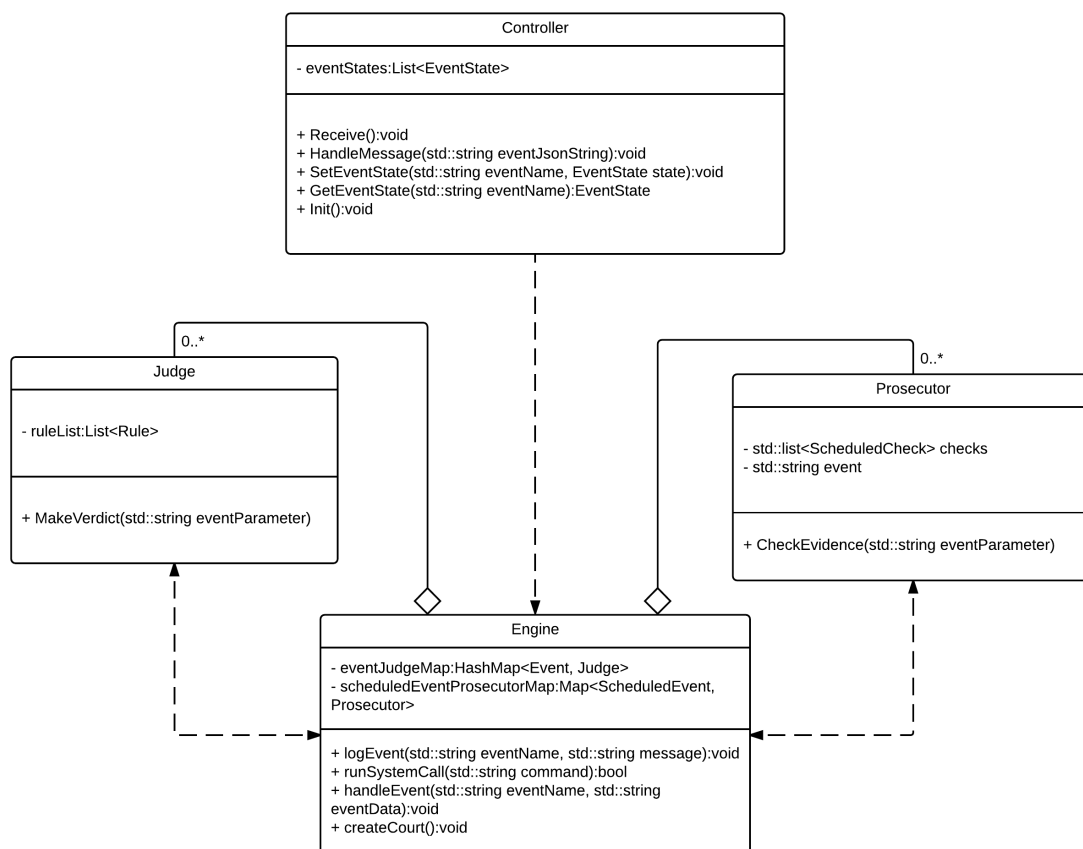


Figur 4.2: Diagram iteration 2

4.3.10 Tredje och slutgiltiga iterationen

I tredje iterationen flyttades fokus från den täta kopplingen mellan Monitor-objekt och Controller-objekt. Genom att istället fokusera på att vilken process som helst kan nyttja Controller-objektet för att aktivera en händelse som i sin tur är knuten till ett Judge-objekt möjliggjordes en mycket generell applikation.

Tredje iterationen introducerade även konceptet Prosecutor vars uppgift var att hantera schemalagda händelser. Schemalagda händelser är händelser som automatiskt försöker aktiveras vid specifika intervall. Det finns även möjlighet att lägga Checkar på dessa händelser. Checkarna består av kedjor liknande de regler som Judge-objekten hanterar. Ett Prosecutor-objekt tar emot en schemalagd händelse och går igenom de Checkar associerade med den schemalagda händelsen. Skulle någon av checkarna misslyckas går Prosecutor-objekten direkt till Controller-objektet och aktiverar den associerade händelsen.



Figur 4.3: Diagram iteration 3

4.3.11 Jämförelse med klassisk EDA och BRE

Delar inom EDA

En klassisk händelsemotor, en EDA, innehåller fyra separerade delar. *Generatorer* som genererar händelser, *kanaler* där händelser transporteras, *bearbetning* där händelse evalueras och slutligen *aktiviteter* som är följder för en händelse.

I projektets händelsemotor motsvaras en *generator* av en Prosecutor, men också av vilken applikation som helst som kan prata med den Socket händelsemotorn lyssnar på. *Kanalen* är i projektets fall den UNIX-Socket som sätts upp vid initiering av systemet. Motorns *bearbetning* sker i Judge-objekt, och *aktiviteterna* är de skript, eller kommandon, som exekveras av ett Judge-objekt.

4.3.12 Skillnader mot klassisk BRE

En klassisk BRE är tätt ihopkopplad med en större databas som den evaluerar sina regler mot. Projektets händelsemotor hanterar istället data i realtid och aktiverar associerade beteenden efter evaluering.

4.4 Front-end

Front- och back-end skall ej vara beroende av varandra. Genom att bygga separata processer kan då framförallt back-end processen placeras som verktyg inuti många tjänster. Detta nås genom att ha en gemensam nämnare, i projektets fall möjliggör en databas enkel hantering av den data som skall ligga till grund för de processer som körs.

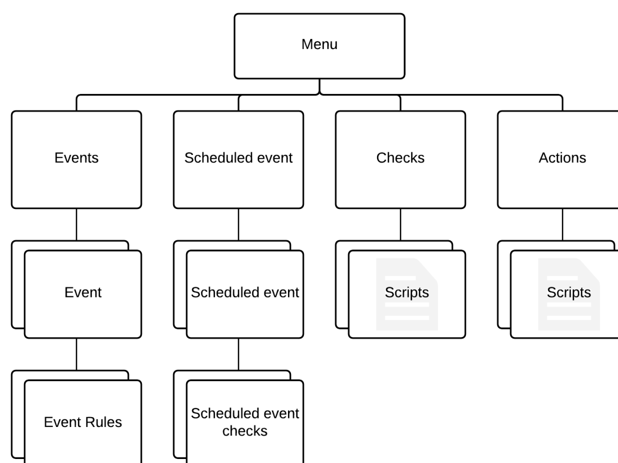
Händelsemotorns front-end implementerades som ett web-gränssnitt. Målsättningen var att uppnå ett intuitivt gränssnitt där men enkelt kunde manipulera den data som användes för att skapa de händelser och regelkedjor som var påtänkta att användas. Hur delarna implementerades presenteras i detta avsnitt.

4.4.1 Användarvänlighet

Baserat på tanken kring att business rules skall kunna vara skrivna av nästan vem som helst [4] har mycket planering lagts på att slutresultatet skall vara så enkelt som möjligt för slutanvändaren att hantera. Att utvecklarna inte skall behöva göra alla ändringar utan att fler personer skall kunna bidraga till kedjan av händelser.

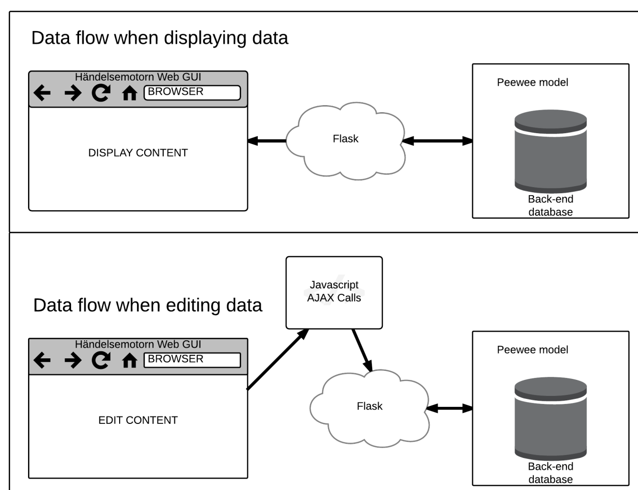
4.4.2 Grafiskt web-gränssnit

För att skapa den web-back-end som behövdes används Python tillsammans med Flask och Peewee. Flask hanterar HTTP-anrop och kallar sedan på databasmodellen implementerad med hjälp av Peewee. Flask tillgodoser även användaren med den grafik och HTML som användarens webläsare behöver för att presentera innehållet. Hur strukturen av webgränssnittet ser ut finns beskrivet i figur 4.4.



Figur 4.4: Diagram över projektets web-gränssnitt

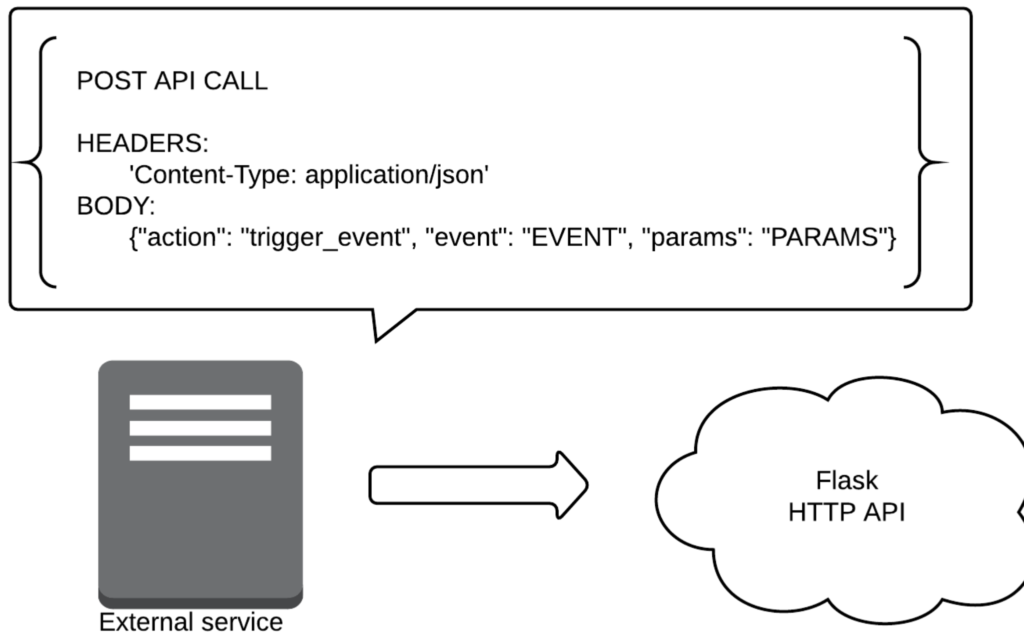
Genom att skapa mallar i HTML för de olika sidorna och presentera dem med data hämtad via Peewees databasmodell kunde man se och ändra aktuell data i databasen. Hanteringen av data gjordes senare via anrop til l javascript-funktioner som i sin tur pratade via HTTP med Flask. Då Flask mottog datan anropades databasmodellen skapad genom Peewees som i sin tur sparade ner datan i SQLite databasen. Se figur 4.5 för en överblick hur det fungerar.



Figur 4.5: Data flöde från web-gränssnitt till Flask

4.4.3 HTTP-API

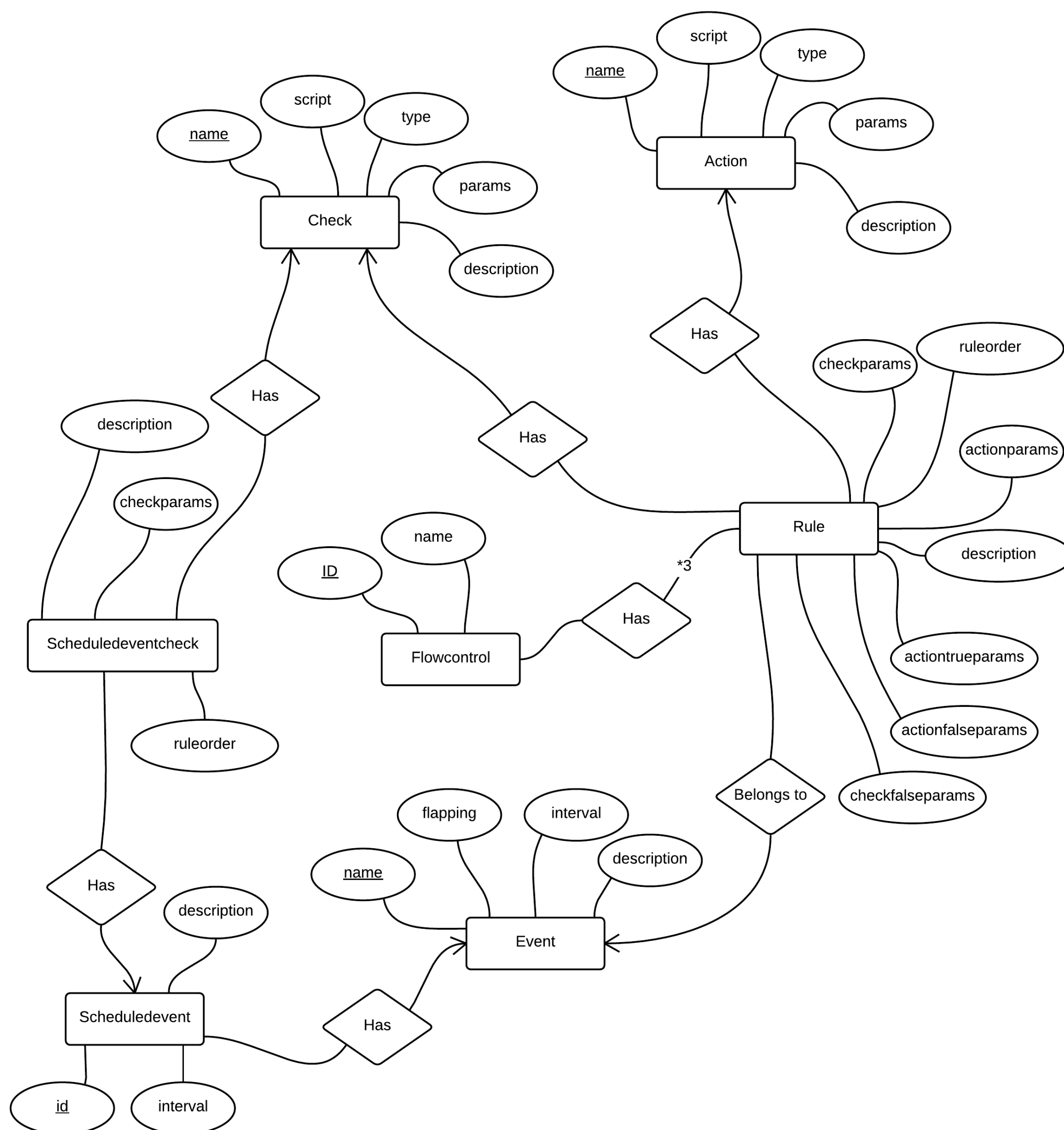
För att kunna aktivera händelser från externa tjänster skapades ett HTTP-API. Man kunde genom detta aktivera händelser skapade i det grafiska web-gränssnittet via HTTP anrop. Detta uppnåddes genom att knyta en URL där man genom ett POST anrop kunde aktivera en händelse. Se figur 4.6 för en överblick.



Figur 4.6: API anrop från extern tjänst

4.4.4 Databas

Databasen som användes under utvecklingen är en SQLite-databas. Den erbjuder ett strukturerat sätt att hantera lokal data via ett abstraktionslager baserat på SQL. Genom att använda en SQL-databas underlättas även skapandet av olika front-ends då SQL är ett vanligt förekommande verktyg för att hantera data. Projektets databasmodell finns beskriven som figur 4.7.



Figur 4.7: ER-diagram över projektets databas

4.5 Tester

4.5.1 Automatiserade tester

Google Test

För att testa C++-kod används Google Test. Testerna går igenom all grundläggande funktionalitet för händelsehanteringen. Mer specifikt så går testerna igenom all funktionalitet i Judge-, Prosecutor- och Controller-klasserna. Det som ännu är otestat är det som benämns som ”Engine” i systemet. Det vill säga den delen som tar emot händelser utifrån och medlar mellan de ovan nämnda klasserna.

4.5.2 Funktionella tester

Testerna är funktionella, det vill säga, de kallar på klassernas metoder under förutbestämda förhållanden, och kontrollerar efteråt att det som skall ha hänt har hänt. Till exempel kan man skapa en regelkedja som säger att en temporär fil ska skapas i filsystemet under vissa förutsättningar. Om man då skapar dessa förutsättningar och säger till ett Judge-objekt att evaluera händelsen, kan man i efterhand kontrollera att filen existerar eller inte existerar, beroende på de förutsättningar man satte upp.

4.6 Integrering med befintliga system

Integrering med händelsemotorn kan ske på tre olika sätt. Anrop från externa system kan använda det HTTP-API som är tätt sammankopplat med det web-gränssnitt som utvecklats. Då systemet är byggt för att även fungera utan front-end finns det möjlighet för externa applikationer att kommunicera med motorn via den UNIX-socket som Controller-objektet lyssnar på.

Det tredje sättet att integrera externa system är att skriva skript som i sin tur kommunicerar med de tredjeparts program man önskar integrera med. De kan kopplas till en händelse. Händelsen kan vara schemalagda eller aktiveras från annat håll.

5
Resultat

5.1 Förundersökning

Liknande motorer

Projektet har ej lyckats med att hitta liknande motorer som utför det som var efterfrågat i bakgrund och mål.

Regelsyntax

Projektets regelsyntax är följande ”IF <check> DO <action> ? <flowcontrol> : <flowcontrol> ELSE <flowcontrol>” och är i projektets mening ett effektivt regelsyntax.

5.2 Utveckling

Back-end

Projektet har lyckats med att undersöka befintliga lösningar och teorier i form av BRE och EDA. Det finns även en konkret och väldokumenterad design och en stabil vältestad och fungerande prototyp skriven i C++. Det genomfördes även en undersökning om möjliga problem beträffande exekvering av externa kommandon från en C++-miljö. Programvaran arbetar mot en databas vilket möjliggör dynamisk hantering av den data händelser och regler innehåller.

Front-end

Den slutliga versionen av web-gränssnittet har möjlighet att skapa nya händelser samt lägga till regler och följder för dessa regler. Man kan även dynamiskt skapa nya Checks och Actions vilket man sedan kan addera till en händelsekedja. Det finns möjlighet att schemalägga händelser med och utan andra Checkar, vilket möjliggör en funktionalitet att kontrollera olika system och processer på ett effektivt sätt. Inuti web-gränssnittet finns en textredigerare anpassad för kod, vilket underlättar framtagning av Checks och Actions. Tjänsten stöder även idag ett HTTP-api som kan aktivera de händelser man skapat. Datan hanterad av web-gränssnittet sparas i samma databas som back-end processen använder.

Central version

Den back-end tillsammans med det web-gränssnitt som utvecklats under projektets gång är anpassat för att användas som en centralt belägen version av programmet. Systemet har under projektets gång installerats i enkla miljöer men blev mot slutet av projektet placerat i en ordentlig produktionsmiljö.

6
Slutsats

6.1 Användarvänlighet

Mycket fokus i projektet har varit att få slutprodukten så enkel och användarvänlig som möjligt. Genom att flytta mycket av komplexiteten internt och skapa rätt förutsättningar kring hur en slutanvändare enkelt kan använda och påverka flödet i Händelsemotorn tror vi att detta är en nyckeln till att systemet skall bli välanvänt och omtyckt.

Om man har som mål att implementera ett system som Händelsemotorn i tjänster där inte alla användare är utvecklare är det viktigt att man utformar en tjänst som är väldigt enkel att använda.

6.2 Fortsatt utveckling

6.2.1 Front-end

En av de största utmaningarna med att bygga en händelsemotor är att göra det användbart för en stor användarbas. Det är lättare att mäta uppnådd funktionalitet i back-end än det är att uppskatta användarvänlighet i front-end. Något som behöver utvecklas vidare är web-gränssnittet så att nya användare enklare kan skapa egna händelser efter sina egna önskemål. Då målet även är att integrera händelsemotorn i andra befintliga system än bara en centraliserad variant behöver man även fundera över hur man skapar en enkel hantering av händelser och regler då man ej har tillgång till det web-gränssnitt som utformats i detta projekt.

6.2.2 Back-end

Säkerhet

Under utvecklingen har mest fokus lagts på den grundläggande funktionaliteten. Något som förbisetts i prototyputvecklingen är säkerhet kopplad till dynamisk skriptexekvering. Idag har man möjlighet att köra vilka skript som helst, vilket inte skulle göra det svårt för någon med lite programmeringskunskap att förstöra mycket i det befintliga systemet.

För att en färdig version skulle kunna användas i tjänster behöver man analysera säkerhetsaspekter, utforma kontroller så att de skript som skrivs ej utför något som skadar systemet där händelsemotorn är installerad, samt skapa policys över vem

som får göra vad. Man behöver även fundera över vilka skriptspråk som ska kunna användas och hur man kan begränsa dem.

Skriptexekvering

Då exekveringsundersökningen som genomfördes under förstudien av händelsemotorn visade en stor skillnad i exekveringstid mellan olika metoder, är det viktigt att vidareutveckla hur motorn hanterar sina skript. Att sekventiellt exekvera ett simpelt skript, det vill säga ett skript som i sig självt inte kräver noterbara tidsresurser, tusen gånger tar via systemanrop runt tolv sekunder samtidigt som en inbyggd tolk exekverar samma arbetsmängd på under en halv sekund.

Om användningen av händelsemotorn blir såpass omfattande att den behöver exekvera ett hundratal skript per sekund, blir det i längden ohållbart att exekvera skripten via systemanrop.

UNIX-Socket/MQ

Motorn använder sig i dagsläget av en enkel UNIX-socket då den tar emot meddelanden. Denna metod har en del brister i det att meddelanden ej kan skickas till motorn om en Controller inte är igång. Skulle systemet istället byta till att nyttja en 'Message Queue (MQ)' skulle denna existera separat från både händelsemotorn och externa parter. Om ett meddelande skulle skickas till en MQ då inget Controller-objekt aktivt lyssnar sparas meddelandet tills det att motorn startas igen. En MQ har även inbyggt stöd för prioritering av meddelanden. Denna funktionalitet saknas helt i projektets framtagna motor.

6.3 Slutdiskussion

Vår rapport tar upp ämnet dynamisk skriptexekvering där det i dagsläget inte finns några fullt tillfredsställande lösningar. I alla fall inte då man vill låta valet av skriptspråk vara fritt.

Området händelsehantering är väldigt omfattande. Det finns en del kommersiella lösningar, men de flesta företag verkar implementera egna system. Vårt arbete kan förhoppningsvis användas som språngbräda för utveckling av mer specifika händelsemotorer för särskilda intressen.

6.3.1 Hållbar Utveckling

Händelsemotorn kan användas för att lätta arbetsbördan för många medarbetare, bland annat supportpersonal, på Icomera. Mycket av det arbete som idag utförs manuellt går att enkelt implementera och styra i den framtagna motorn. Detta leder till en trevligare och mindre påfrestande arbetsmiljö för många anställda på företaget.

Litteraturförteckning

[1] MQ_OVERVIEW(7), 2016.

[2] Unix(7) linux programmer's manual, 2016-03-15.

[3] Malcolm Chisholm. *How to Build a Business Rules Engine*. Morgan Kaufmann, 2004.

[4] Malcolm Chisholm and Malcolm Chisholm. 2 – Why Build a Business Rules Engine? In *How to Build a Business Rules Engine*, pages 9–19. 2004.

[5] Allan Kolber et al. Defining business rules. Project report, the Business Rules Group, 2000.

[6] Michael Kerrisk. *The Linux Programming Interface*. 2010.

[7] Charles Leifer. Peewee ORM.

[8] BM Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2006.

[9] Armin Ronacher. Flask.

[10] William Richard Stevens. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*. 1999.

[11] Wikipedia. Pair programming, maj 2016.

[12] Wikipedia. SQL, maj 2016.