Felix Blix Everberg & Hakon Primdahl -- Aarhus School of Engineering

ITTAPK -- 24/10/16 -- Søren Hansen

Cargo Management System
A Train Station Simulation

Introduktion

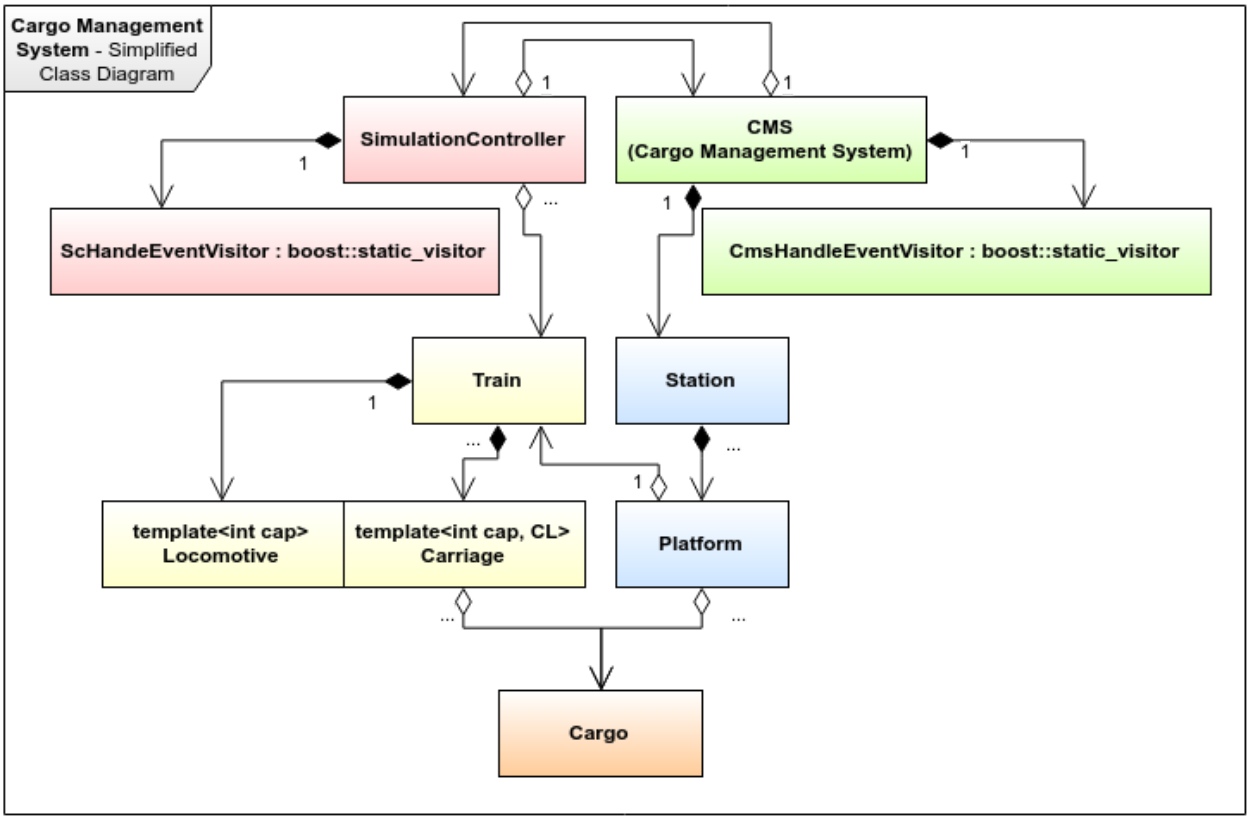
Denne rapport beskriver det projekt der skal danne grundlag for eksamen i ITTAPK efteråret 16.
Målet med projektet har været at bruge de teknikker og faglige elementer af C++ og det afskygninger der er blevet gennemgået i undervisningen, for at vise en generel forståelse af fagets indhold. De fleste dele af koden er skrevet med henblik på at forøge kvalitet/udførsel af kode, men nogle elementer er dog blevet inddraget blot for at vise forståelse for et større omfang af undervisningsmaterialet.

Projekt/System Beskrivelse

Dette system går i al sin enkelthed ud på at simulere et system der kan håndtere lastningen af et vilkårligt antal tog, med en arbitrær mængde gods. Hvad det vil sige er, at vi har to tråde ud over main tråden der hver håndterer deres del af programmet. Den ene tråd eksekverer *SimulationController* som starter en simulering af tog der kører til et *Cargo Management System* som har en station med perronner som indeholder en arbitrær mængde forskelligt gods hvor togene kan blive lastet og sendt tilbage til *Simulation-Controller*. Simuleringen kører indtil alle perroner er tomme for gods, eller til togene ikke kan hente mere, fordi ingen af togenen har den rette type laste egenskab for det tilbageværende gods på perronerne. Alle tog bliver tømt hver gang kontrollen af dem returneres til *SimulationController*.

System Arkitektur

Nedenstående er et simplificeret klasse diagram, herefter følger en beskrivelse af system arkiteturen. Et detaljeret klasse diagram findes som bilag i PDF form.(CMS Class Diagram.pdf)



SimulationController

Indeholder en reference til en liste af de tog der bruges i simuleringen, en reference til det CMS som bruges i simuleringen, samt en ScHandleEventVisitor til håndtering af de forskellige events der forekommer i simulation controller.

CMS (Cargo Management System)

CMS er den klasse der håndterer togene så snart der er sendt signal om at de er ankommet til stationen. CMS indeholder en station, en reference til en SimulationController og en CmsHandleEventVisitor til håndtering af events i cms.

trains.hpp - Train / Locomotive & Carriage

Filen trains.hpp indeholder tre forskellige klasser. Den første af de tre er Train, som implementerer vores interface udadtil, den har abstrakte funktioner som kan laste of aflaste toget, tjekke kapacitet og hvad den kan lastes med. Hvad den kan lastes med afgøres af hvilke slags carriages den har med i sin Carriage liste. Carriage er den første af de to template klasser i trains.hpp. Den bruges i trains som en Carriage liste, hvor hver carriage templatiseres ved en kapacitet og en liste af cargo specialiseringer. Forskellige carriages kan fragte forskellige ting. Der er følgende to begrænsninger for hvilke cargo specialiseringer en Carriage kan have: En vogn der transporterer væsker kan kun transportere væsker, og en vogn der transporterer husdyr kan kun transportere en type husdyr, men kan godt samtidig transportere regulære godstyper, så længe kapaciteten ikke bliver overskredet. Et locomotive templatiseres blot ved den kapacitet, altså hvor meget vægt den kan trække. Et tog skal have et lokomotiv stort nok til at trække alle dets vogne i fuldt lastet tilstand.

station.hpp - Station / Platform

Station.hpp består af to dele, en Station klasse og en Platform klasse. En station indeholder en liste af platforme med en arbitrær størrelse. En platform har en pointer til et tog, og kan kun håndtere et tog ad gangen. Derudover har platform også en liste af cargo med arbitrær størrelse og typer, instantieret og givet med fra main. Hvis alle platforme er fyldte, bliver togene føjet til en kø, som tømmes når der igen er plads på en platformen. I virkeligheden er køen måske ikke den ideelle måde at implementere det på her, da det ville være smartere at tjekke køen igennem for hvilke tog der passer bedst på platformen inden de bliver tømt ud af køen, men den funktionalitet er blevet ofret for at få tid til at få andet til at virke.

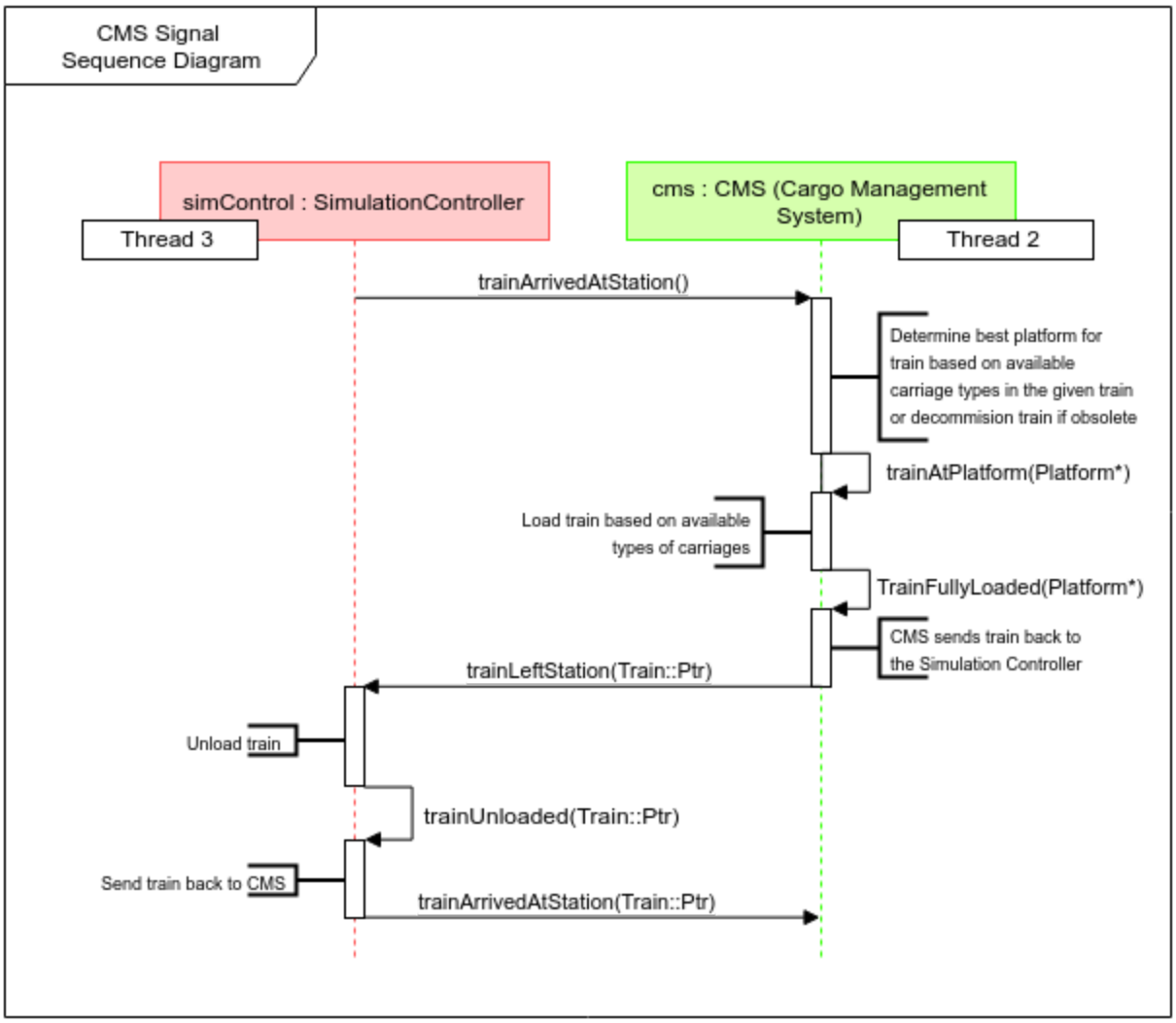
cargo.hpp - Cargo

Cargo er en relativt simpel model klasse der bliver specialiseret til 9 forskellige cargo typer. Hver gang der bliver tilføjet noget Cargo til en vogn, bliver det tjekket om denne har de relevante Cargo specialiseringer til at kunne holde den bestemte type cargo. De

nedarvede typer implementerer to traits, som indikerer om de er hhv. Flydende- eller husdyr-last.

Cargo Management Signaler - Design

Herunder ses et sekvens diagram der beskriver kommunikationen i mellem de to hovedtråde, hver tråd har en event handler som håndterer de forskellige signaler og sørger for at sende et nyt signal når en blok er blevet afviklet. Grunden til cms har fået tildelt thread 2 og simControl thread 3, er at main afvikles på thread 1, dette sker automatisk.



Kommunikation foregår mellem de to hovedtråde foregår som sagt med signals, de signals er implementeret med boost biblioteket signals2, som er boosts implementering

af publisher-subscriber mønstret. De funktioner der er tilkoblet signalerne i de to klasser (CMS og SimulationController) tilføjer et event til sin klasses event kø, klasserne afvikler så disse events parallelt og sørger for at togene bliver sendt rundt indtil de ikke kan hente mere gods. Systemet fungerer sådan at simControl starter med at kontrollere en række tog, som er instantieret til at kunne bære forskellige typer carriage. Hvert tog bliver med et signal overleveret til CMS tråden, som håndterer lastning af de enkelte tog. Hver gang CMS modtager et tog tjekkes det hvilken platform der kan laste mest muligt gods til det, og hvis det ingenting kan laste, bliver toget decommissioned, hvilket vil sige at det bliver flyttet til en liste af ubrugelige tog. Hvis toget bliver lastet med gods, sendes det tilbage til SimulationController. Da det bare er en simulering og destinationen efter CMS er ligegyldig, tømmes togets last ud i ingenting. Når der ikke er mere gods tilbage på stationen, eller alle tog er blevet flyttet til listen med ubrugelige tog, afsluttes simuleringen.

Implementering af signaler

Signals er med for at have en måde at kommunikere i mellem vores to main tråde. Signals fungerer sådan at man erklærer et callable object med som man så connecter til en funktion eller et andet callable object, i dette projekt er alle signaler forbundet til funktioner med lambda udtryk.

```
//Signals ********************************************************
boost::signals2::signal<void(cm::Platform*)> trainAtPlatform;
boost::signals2::signal<void(cm::Platform*)> trainFullyLoaded;
boost::signals2::signal<void(cm::Train::Ptr)> trainLeftStation;
//***************************************************************
```

Øverste kode snippet viser deklareringen af tre signaler fra klassen CMS.

```
    trainAtPlatform.connect([&](cm::Platform *p) {
        PushEvent(Event_TrainAtPlatform(p));
    });
    trainFullyLoaded.connect([&](cm::Platform *p) {
        PushEvent(Event_TrainFullyLoaded(p));
    });
    trainLeftStation.connect([&](cm::Train::Ptr t) {
        PushEvent(Event_TrainLeftStation());
    });
```

Nederste kode snippet viser hvordan vores funktion PushEvent bliver bundet ved hjælp af lambda funktioner til vores signals.

En smart ting ved signals, er at man kan have så mange listeners, eller slots som det kaldes i boost::signals2, som man vil, på et signal, uden at det skaber problemer. Det kunne i dette projekt for eksempel, med små ændringer i koden, bliver benyttet til at tilføje flere stationer som togene kunne fragte gods imellem, eller man kunne lave en visualizer, der viste på en smart måde hvilke signaler der blev kaldt hvornår og hvorfra, så man kunne få et overskueligt interface til observation af systemet.

Boost Variant

I dette projekt spiller boost::variant en vigtig rolle i to sammenhænge: Event systemet i CMS og SimulationController klasserne bruger visitor mønsteret til håndtering af de forskellige event typer, og i TrainImpl konstrueres der automatisk en variant type med tilhørende visitors, til at kunne tilgå de forskellige carriages, som alle har forskellige typer, givet deres templatiseringer.

I event systemerne defineres der for hver type event et struct, med et sigende navn, og data felter for de ting der skal sendes med det givne event:

```
struct Event_TrainAtStation{
    Event_TrainAtStation(Train::Ptr t) : train(t){}
    Train::Ptr train;
};
```

Herefter implementeres en Visitor klasse, som indeholder event håndterings koden for hver event type.

```
class CmsHandleEventVisitor : public boost::static_visitor<>{
public:
    CmsHandleEventVisitor(CMS* cms);
    void operator()(const Event_TrainAtStation& e) const;
    void operator()(const Event_TrainAtPlatform& e) const;
    void operator()(const Event_TrainFullyLoaded& e) const;
    void operator()(const Event_TrainLeftStation& e) const;
    void operator()(const Event_Shutdown& e) const;
private:
    CMS* _cms;
};
```

Disse er i dette system implementerede som nestede klasser i de event drevne klasser, og instantieres så i konstruktoren med en pointer til ejer klassen, således at member funktioner nemt kan kaldes når der håndteres et event. Hvert event kan nu håndteres ved, på en trådsikker måde, at hente ens variant fra event køen og kalde boost::apply_visitor:

```
void cm::CMS::EventHandler() {
    std::unique_lock<std::recursive_mutex> lock(cond_m, std::defer_lock);
    //TODO: add end condition
    while (running) {
        //lock.lock();
        while (eventQueue.empty()) {
            wait(lock);
        }
        lock.lock();
        Event e = eventQueue.front();
        //tp::print("handling event\n");
        eventQueue.pop();
        lock.unlock();
        boost::apply_visitor(event_visitor, e);
        //tp::print("done handling event\n");
    }
}
```

Dette tillader en overskuelig og uniform måde at håndtere events på, og ønskes det at implementere et ekstra event, skal dette blot tilføjes til variant typen, og i visitor klassen.

Den anden brug af variant demonstrerer hvordan man kan tilgå en række klasser med en fælles funktionalitet, men uden en fælles forfader, på en homogen måde. Hvert tog har en række Carriage template klasser, som for demonstrationens skyld ikke nedarver fra en fælles forfader. Alle disse typer kendes ved instantieringen af TrainImpl fra dens CarriageList template parameter, og der bliver således lavet en typedef ved hjælp af structet MAKE_BOOST_VARIANT. Vha. variant typen kan togets vogne så holdes i en std::list container:

```
class TrainImpl : public Train {
public:
    typedef typename MAKE_BOOST_VARIANT<CARRIAGE_L>::VARIANT_TYPE carriagevariant_t;
    typedef std::list<carriagevariant_t> carriagelist_t;
```

Herefter laves der så automatisk en visitor for hver af de nødvendige funktioner ved at lade en visitor klasse nedarve fra sig selv, templatiseret ved typelisten fra Variant typen, således at hvert niveau af den rekursive nedarvning implementerer en af visitorens operator overloads:

```
template<typename CL>
class CanHoldVisitor : public CanHoldVisitor<typename CL::TAIL> {
public:
    CanHoldVisitor(Cargo::Ptr c) : CanHoldVisitor<typename CL::TAIL>(c) {};

    virtual bool operator()(typename CL::HEAD &e) const {
        return e.canHold(cargo);
    }

    using CanHoldVisitor<typename CL::TAIL>::operator();
    using CanHoldVisitor<typename CL::TAIL>::cargo;

};

template<>
class CanHoldVisitor<CL_NULL_ELEM> : public boost::static_visitor<bool> {
public:
    CanHoldVisitor(Cargo::Ptr c) : cargo(c) {}

    CanHoldVisitor(const CanHoldVisitor &other) : cargo(other.cargo) {}

    virtual bool operator()() const { return false; }

    Cargo::Ptr cargo;
};
```

Base case hvor template typen er CL_NULL_ELEM, nedarver så fra boost::static_visitor, og implementerer constructoren som gemmer de nødvændige data for klassen.

Meta Programmering

Som nævnt ovenfor i afsnittet om train.hpp, er vores tog implementeringer templated med en række ting, herunder hvilke vogne de har, og hvilke typer cargo disse can acceptere. Denne templatisering sker på classen TrainImpl, som arver fra den abstrakte klasse Train, således at disse kan behandles homogent vha. Dynamisk polymorphisme.

Konstruktionen af tog er pålagt nogle begrænsninger, som alle tjekkes ved compiletime gennem SFINAE paradigmet. TrainImpl templatiseres ved hhv. En Locomotive type og en CarriageList som er en typelist begrænset til Carriages. Første begrænsning er at et togs vogne ikke tilsammen må kunne holde mere vægt end lokomotivet kan trække. Dette bliver tjekket vha. Static_assert, som også bruges en række andre steder for at spare tid.

```
static const int capacity = CAPACITY_SUM<CARRIAGE_L>::value;
static_assert(LOCOMOTIVE::capacity >= capacity,
              "The constructed train does not have a sufficiently strong locomotive to pull all it's carriages.");
```

Hver enkelt Carriage er templatiseret ved en kapacitet, som er den der summeres i CAPACITY__SUM, og en CargoList, som er en typelist begrænset til underklasser af Cargo.

Her er der skrevet to assertions uden brug af static_assert, og dermed i en implementering der kunne have været brugt før c++11, idet hver Carriage instantierer structet IS_A_VALID_CARGO_LIST templatiseret ved dens cargo liste:

```
typedef IS_A_VALID_CARGO_LIST<CL> META_INFO;

const bool isTanker = META_INFO::IS_TANKER::value;
const bool hasLivestock = META_INFO::HAS_LIVESTOCK::value;
```

Dette struct kan kun instantieres såfremt den givne CargoList opfylder de følgende to regler: Indeholder listen et element af flydende cargo, så må den kun indeholde flydende cargo typer, og indeholder den en livestock type, må den ikke indeholde andre livestock typer.

Disse to begrænsninger er lavet som hver deres struct, pakket ind i structet IS_A_VALID_CARRIAGE_LIST:

```
template<typename CL>
struct IS_A_VALID_CARGO_LIST {
    //typedef ASSERT_IS_CARGO<CL> ACCEPTS_CARGO;
    typedef ASSERT_TANKER_VALIDITY<CL> IS_TANKER;
    typedef ASSERT_LIVESTOCK_VALIDITY<CL> HAS_LIVESTOCK;
};
```

ASSERT_TANKER_VALIDITY kan kun instantieres hvis CL opfylder den første betingelse, og har efterfølgende feltet value, som indeholder hvorvidt den givne liste indeholder væsker eller ej. Den er i virkeligheden en wrapper for et andet struct som hedder ALL_OR_NONE_ARE_LIQUID, og de er implementeret som følger:

```
template<bool, typename E, typename L>
struct ALL_OR_NONE_ARE_LIQUID;
template<typename E, typename L>
struct ALL_OR_NONE_ARE_LIQUID<true, E, L> {
    static const bool value = ALL_OR_NONE_ARE_LIQUID<
            IS_LIQUID_CARGO<E>::value == IS_LIQUID_CARGO<typename L::HEAD>::value,
            typename L::HEAD, typename L::TAIL>::value;
};
template<typename E>
struct ALL_OR_NONE_ARE_LIQUID<true, E, CL_NULL_ELEM> {
    static const bool value = IS_LIQUID_CARGO<E>::value;
};
template<typename CL>
struct ASSERT_TANKER_VALIDITY {
    static const bool value = ALL_OR_NONE_ARE_LIQUID<true, typename CL::HEAD, typename CL::TAIL>::value;
};
```

Den primære template er ikke implementeret, men der er to specialiseringer. I begge disse er det første template argument true, hvilket er hvad der bruges til at tvinge en substitution error. De to andre parametre bruges til rekursivt at pakke type listen ud. Første specialisering er den rekursive case. Her tjekkes det om de første to elementer enten begge er flydende, eller ej vha. IS_LIQUID_CARGO, ved at assigne value til value fra structet selv, kaldt med resultatet af tjekket og resten af listen. Dette fortsætter så indtil base case implementationen bliver ramt, når næste element er CL_NULL_ELEM typen der bruges til at slutte CargoList og CarriageList. I dette case sættes value feltet til true hvis sidste element er flydende, idet de alle må være ens for at base case overhovedet bliver instantieret.

ASSERT_LIVESTOCK_VALIDITY er implementeret på en lignende måde ved brug af bools, og rekursiv instantiering, men vil ikke blive gennemgået her.

Hjælpe meta funktionen IS_LIQUID_CARGO, fungerer på samme måde som IS_LIVESTOCK_CARGO, ved at tjekke om dens template type typedef er boost::mpl::true_ som IS_LIQUID. I Cargo base klassen defineres begge traits til false, og enhver klasse der nedarver fra Cargo kan så omdefinere det som det ses nedenfor i Water klassen:

```
class Water: public Cargo{
public:
    typedef boost::mpl::bool_<true> IS_LIQUID;
    Water(int w): Cargo(w, LIQUID_LOAD_FACTOR*w){}
    ~Water(){};
};
```

Cargo.hpp, og Trains.hpp indeholder flere template baserede metaprogrammerings elementer, men de ovennævnte repræsenterer udmærket de benyttede koncepter.

Diskussion

Der har været mange overvejelser og en del ændringer undervejs i projektet, især implementeringen af hvordan signals bliver brugt er blevet ændret og forbedret. Grundet signals natur og den måde vi benytter os af det på, er event handleren blevet tilføjet, hvor der før kun blev kommunikeret både imellem klasser og internt gennem signals. Det er ikke en specielt optimal måde at håndtere kommunikationen på, da signals egner sig bedre til at blive brugt i et pub-sub system hvor der enten er flere publisher og/eller

subscribers, så derfor blev event køen tilføjet for at efterkomme nogle af de problemer der opstod ved en ren signals implementering.

Konklusion

I dette projekt har vi fået demonstreret en række af fagets koncepter, og hvordan disse kan tilbyde enten en simplere implementering, eller funktionalitet der ikke ville kunne have ladet sig gøre uden deres brug. Ud gennem projektet benyttes ting som STL containers, nogle udvalgte algoritmer, C++11 syntax, og C++11's tråd interface, til at simplificere implementeringen.

Meta programmering tillader komplekse typetjæk ved kompiletime, uden nogen indvirkning på eksekverings tid, boost variant tillader homogen tilgang til objekter uden brug af polymorphi, og boost signals giver en fuld implementation af publisher subscriber modellen, som vi bruger til at interface vores tråddede klasser.

Nogle features vi ikke har haft i betragtning i projektet er exception safety, custom containers and itterators, boost_statechart, og de forskellige betragtninger drøftet i relation til embedded C++. Derudover er der en del forbedringer af koden som der aldrig blev tid til, samt en fejl der kan lede til at simuleringen fortsætter uendeligt, som vi ikke nåede til bunds i. Disse ting er dog ikke større end at de ikke kommer i vejen for illustrationen af de andre koncepter.

Build Instructions

For at bygge koden kan man benytte CMake ved hjælp af følgende komandoer fra roden af det leverede kode:

```
mkdir build
cd build
cmake ..
Make
```

Herefter kan koden køres således:

```
./CMS
```