NTNU – Trondheim
Norwegian University of
Science and Technology

Building Intelligent Transport Systems with Reactive Blocks and OSGi

Eivind Sandstad

Master of Science in Communication Technology
Submission date: June 2014
Supervisor: Frank Alexander Krämer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Intelligent transport systems enable applications within public and private transport. Making it possible for vehicles to communicate with each other and roadside installations. It also makes for the possibility of making the traditional equipment, such as traffic lights, smarter by providing them with more data to base their behavior on. Within this task, an ITS application should be built using Reactive Blocks and OSGi. The overall goal of the project is to research and report the potential benefits of using Reactive Blocks and OSGi in ITS applications.

Abstract

Traffic systems are the cause of both significant carbon emissions and injuries in our world today. In an effort to better the traffic systems, a large amount of work is being put into making them intelligent. By increased information sharing and decision-making based on better data, intelligent traffic systems (ITS) hope to increase efficiency and safety on the roads. Making ITS can be quite complex as the systems are already complex and large, it is therefore important to make the development and quality of ITS as good as possible.

This thesis aims to find out whether Reactive Blocks and OSGi, a modelling based tool and a component system run in Java, is the right platform on which to build an ITS, specifically in the Norwegian traffic system environment. Simply put, the question the thesis aims to answer is: To what degree is Reactive Blocks and OSGi a good platform for ITS development?

To answer the aforementioned question, a literature study has been conducted, as well as making and testing a prototype application. From the results of the literature study and the prototype a theoretical evaluation of the platform has been made.

The results indicate that Reactive Blocks and OSGi is a very good fit to the platform of ITS. It is in all likelihood a right choice when developing ITS. It has beneficial features that the competition lacks, that make up for its respective drawbacks.

Based on the results, Reactive Blocks and OSGi is recommended for developing ITSs.

Sammendrag

Trafikksystemer står for både en signifikant del av klimautslipp og skader på verdensbasis. For å forbedre trafikksystemene jobbes det med å gjøre dem smarte. Ved å øke informasjonsdeling og å tilføre bedre data til styringssystemer, håper man at intelligente trafikksystemer (ITS) kan øke både effektivitet og sikkerhet på veiene. Siden trafikksystemer allerede er store komplekse samtidssystemer kan det å lage et ITS være veldig komplekst. For å møte så få vegger som mulig er det derfor viktig at man bruker en utviklingsplattform som passer til systemet og gir mest mulig støtte i utviklingsprosessen.

Denne oppgaven gjør et forsøk på å finne ut om Reactive Blocks og OSGi, et modellbasert verktøy og et komponentsystem som bygger på Java, er den rette plattformen for ITS-bygging, spesifikt i Norge. Kort sagt er målet til oppgaven å finne ut hvorvidt Reactive Blocks og OSGi er en god plattform for ITS-utvikling, og i hvor stor grad.

For å finne ut av dette har det blitt gjort en litteraturstudie, og en prototype er laget og testet. Basert på resultatene fra litteraturstudien og prototypen er det gjort en teoretisk evaluering av plattformen.

Resultatene i oppgaven peker mot at Reactive Blocks og OSGi passer veldig godt for ITS-utvikling. Det er i høy grad sannsynlig at Reactive Blocks og OSGi er et rett valg for ITS-utvikling i Norge. Platformen har fordeler som mangler hos konkurrentene, som mer enn gjør opp for ulempene.

Basert på resultatene funnet i oppgaven kan Reactive Blocks og OSGi anbefales for utvikling av ITS.

Preface

This thesis is written as a conclusion to my 5-year MSc in Communication Technology at the Norwegian University of Science and Technology (NTNU). The thesis is the contents of the 10th and final semester (spring of 2014) of my MSc, awarding 30 ECTS credits. It is focused on the process of deducing whether Reactive Blocks and OSGi is well suited for building intelligent transport systems (ITS).

The process has given me an insight into the world of ITS, and the details of Reactive Blocks and OSGi, all of which I have found to be very interesting. Hopefully, some of the same insights will be offered to the reader, sparking a similar interest.

I would like to thank my professor Frank Alexander Kraemer for his assistance and insights during the work, it has been of great help. In addition to that I've found a lot of value in communications with Eric Olsen at Statens Vegvesen and Jo Skjermo at SINTEF, who helped me get an understanding of the world of ITS.

I would also like to thank my parents, Helle Frisak Sem and Olav Sandstad, who were kind enough to read through my thesis, and offer pointers from an outsiders point of view.

Eivind Sandstad

Contents

List of Figures

Chapter 1
Introduction

Technology is getting integrated in more and more of the world we live in. The increased information communication is making our surroundings smart. One of the fields that stand to benefit greatly on information communication, both in efficiency and safety, are the traffic systems. Statens Vegvesen, the state-owned organization that manages the road network in Norway, is currently in the starting phase of transitioning their traffic system into an intelligent traffic system (ITS). Though some of the technologies of the ITS to some extent already is decided, the software development platform is not.

1.1 Problem Description and Scope

Does Reactive Blocks with OSGi, a modelling based tool and a component system based on Java, provide a good platform for intelligent transport system (ITS) development? This thesis aims to answer exactly that. Finding the right platform on which to build a solution can save a great amount of work later in the process. Technologies that don't fit well to the system they are used to design can not only bring more complexity than needed, but can also make avoiding errors very difficult. Designing concurrent systems are, for example, notoriously difficult to get right without being extremely aware of the possible pitfalls. This can be almost completely remedied by the use of a good runtime system and a state machine structure. Seeing as an ITS will almost certainly have to have concurrency, the platform choice matters.

With this in mind the thesis aims to figure out whether Reactive Blocks with OSGi, a relatively new development platform, is a good platform for ITS-systems. Reactive Blocks with OSGi is a graphical modelling based tooling to create robust, concurrent and scalable systems. There are many benefits to the platform, and it is on the cutting edge of new development tools.

1.1.1 Scope

The scope of the thesis is strictly in the field of ITS. The performance and benefits of using Reactive Blocks with OSGi are considered only in this field. Implications on how well the platform works in general may be deduced, but the focus, and aim of the thesis is specific to ITS.

1.2 Structure of the Thesis

This thesis reaches its conclusion based on a combination of the evaluation of a prototype, literature study, and theoretical evaluation. The structure of the theses is a follows:

1.2.1 Technologies

Chapter **??** introduces the technologies that are in focus in the research question. the ultimate goal of the thesis is to evaluate the technologies introduced in the chapter.

1.2.2 Current ITS at Statens Vegvesen

Chapter **??** explores the status quo of ITS development at Statens Vegvesen's ITS project. It aims to bring a better understanding of the situation as it is, so as to get a grasp on what is the best step forwards.

1.2.3 Challenges in ITS

Chapter **??** describes the challenges of building an ITS. The chapter bases its evaluation of ITS challenges upon information from Statens Vegvesen, both technology specifications and the interviews with ITS workers from Statens Vegvesen and SINTEF.

1.2.4 Building ITS-blocks with Reactive Blocks and OSGi

Chapter **??** aims to find the capability of the technologies explored in chapter **??** to make ITS systems. The chapter includes the description and analysis of a prototype.

1.2.5 Alternative Technologies

Chapter **??** presents a set of rivalling technologies that would also be strong choices for building ITS. The aim of the chapter is to examine the competition, to get an understanding of where the main technologies are lacking and exceeding compared to the state of the art.

1.2.6 Discussion

Chapter **??** analyses the results found in chapter **??** with respects to meeting the challenges in chapter **??**, and how the technologies in chapter **??** measures up to the rivalling technologies of chapter **??**.

1.2.7 Conclusion and Further Work

Chapter **??** attempts to provide an answer to the problem description, based on the analysis in chapter **??**. It also suggests some further works to build upon the work done in the thesis.

Chapter 2
Technologies

This chapter consists of a small introduction to the technologies used later in the thesis. The focus is mostly on the general layout of the technologies and their respective benefits.

2.1 Intelligent Transport Systems

Intelligent transport systems (ITS) are all systems for communications between vehicles (car-to-car), and between vehicles and ITS-stations (car-to-infrastructure). ITS also cover ICT for rail, water and air transport, including navigation systems[**?**]. In an ITS data communications would be much more widespread than that of the situation today. In Figure **??** we can see the communication channels that could be utilized for traffic systems for improved traffic management. In today's society, with the amount of time and resources that go into transportation of both people and goods there is an enormous potential for improvement. Not only is transportation costly, it is also dangerous. According to the World Health Organization road injury is responsible for 2.2 percent of deaths world wide [**?**], and that's not counting the effects of the CO2 emissions, which for cars are 33 percent of the total emissions in the US [**?**]. This means that even small improvements in the traffic systems we are currently using could potentially be worth a lot. And this is where ITS comes in, by gathering and using data from the road, and adding communication between the entities in the traffic system there is a huge potential for improvements both in safety and efficiency. This can be anything from improving the traffic flow by controlling traffic lights smarter to temporarily closing unsafe roads based on road friction and wind speed measurements.

Figure 2.1: Visualization of ITS, in this traffic system all the entities are connected in a wireless manner. Image from [**?**]

The practical part of this project will work with road transport (cars), though implications might be made for ITS covering other types of transport.

2.2 OSGi

The Open Source Gateway initiative (OSGi) technology is a dynamic component system for Java. OSGi enables a development model where applications are composed, dynamically, of many different reusable components [**?**]. The goal of the specifications described in OSGi is to enable components to hide their inner workings, so that the implementation of a subsystem does not matter to the rest of the system, and only communicate through OSGi services that share only what needs to be shared. It is hard to be wrong about things you have no knowledge about and make no assumptions of[**?**]. And following this principle, OSGi aims to make systems more scalable, and decrease complexity. There have been talk of component systems for a while in the JVM-community, but the most prominent, and successful of these has thus far been OSGi, as testified by its use in large scale applications such as Eclipse and Spring.

2.2.1 The OSGi Architecture

OSGi follows a layered architecture model, consisting of modules and services. The modules is what carries the modularity of OSGi. Simply put, modularity in this context is not sharing. This is to some extent what the private and public declarations in Java are meant for, though they are a lot harder to use correctly. In OSGi a unit of modularity is referred to as a Bundle. The service model of the architecture provides the functionality that is usually handled by factories in Java. Though in a slightly different way. The way to communicate between bundles is to use services, by the use of a service registry, shown in Figure **??**. Bundles create objects and register them with the service registry, where other bundles can get the services registered. Bundles can listen for types of services and get them once they are registered. This also means that bundles can be switched while registering similar services so that bundles can swap and update without it killing the rest of the system.[**?**]

Figure 2.2: The layered model of the OSGi architecture [**?**]

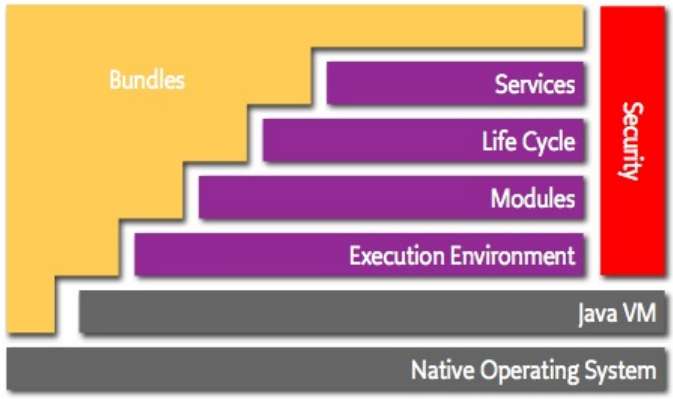
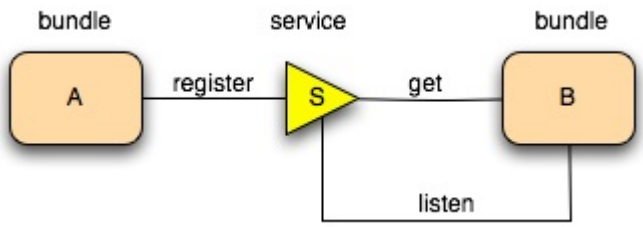


Figure 2.3: The OSGi service model. A bundle can register services, and listen for service types. [**?**]



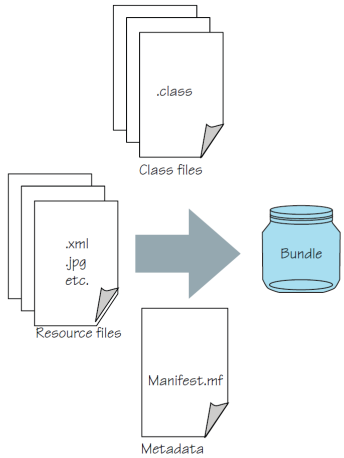
2.2.2 Bundles

The physical manifestation of an OSGi component is called a bundle, and it manifests in the form of a JAR file. These are the meat of any OSGi-application. Though a bundle might seem like any Jar file, it has some subtle details that makes it different. Bundles have essential OSGi metadata in their manifest, detailing the bundle version, the packages within the bundle that should be visible outwards and what packages need be imported. Using this information the OSGi framework does not need to load all the classes in every bundle to find the classes it needs, as is the norm when using normal JARs, adding a great deal to the speed of the system.

Definition of a Bundle

"A physical unit of modularity in the form of a JAR file containing code, resources, and metadata, where the boundary of the JAR file also serves as the encapsulation boundary for logical modularity at execution time." [**?**]

Figure 2.4: A bundle can contain all the usual artifacts you expect in a standard JAR file. The only major difference is that the manifest file contains information describing the bundle's modular characteristics[**?**].



2.2.3 Services

In OSGi a service is an external resource that does some work. They are in essence implementations of interfaces available through the service registry **??**. This is, in a way, OSGi's response to factories. In Java it is quite common to solve the problems of class sharing in large applications by using factories, and this is by all means a valid solution. The problem is that with factories the implementation is specified by the factory, and there is not a lot of room for customization from the requesters side

on which implementation should be used. This is not the case with services in OSGi, here one can be specific or non-specific as to what kind of implementation is wanted, and the service registry will provide the services that are suited to the request. This opens up the possibility of for example having different implementations for different versions of bundles, and due to the dynamic nature of OSGi these can be added and removed while the system is running without causing any harm.

2.2.4 Service registry

The service registry keeps track of the services in an OSGi application and provides these to the bundles running in the system. The use of services in OSGi follows a listen-model, where bundles registers listeners with the service registry, which updates when services become available. This is what makes dynamic communication between services and bundles possible, and relatively simple at that.

2.2.5 Benefits of OSGi

OSGi has many benefits, but the one that is most important with respect to ITS is its ability to update components without downtime. Bundles can be updated without the application ever having to restart or turn off. In systems that need to be running at any given time, such as traffic lights, this is an argument in favour of OSGi. In addition to this main point of interest OSGi offers a lot of other desirable traits, some of the more important ones, with respect to ITS follows. [**?**]

Reduced Complexity

The division into components that focus inwards rather than on the system surrounding them makes subsystems of large and complex systems relatively simple[**?**]. This is a trait of all component systems, and it is often the main reason why such systems are made. With systems that are already of a complex nature, such as road-traffic systems the more simplistic each subsystem needs to be, the better. The keep it simple stupid (KISS) principle is well known throughout the software world, as well as most other markets. The more simple you can make a problem, the faster it will be solved. The relative simplicity of your code also makes software easier to maintain, more flexible, easier to modify and has a smaller entry cost for new developers [**?**].

Reuse

Because OSGi is in the moment of writing the most popular component system, and there is an increasingly large community of open-source projects working with the OSGi technology, there are good opportunities for the reuse of already written applications and components that can be integrated into any system[**?**]. Since OSGi is a component system, the possibility for third-party developed components and

application in the future is also there. So in the case of the ITS application platform being relatively open to outside developers the component structure of OSGi is not a bad structure.

Security

OSGi provides a security model that is an improved version of Java's own, in theory very strong, security model. The problem with the security model used in Java is that it is very hard to configure in the real world. This means that most secure Java applications have two choices: no security, or limited capabilities[**?**]. In OSGi a fine grained security model, like the one in Java, is used, but the usability is improved by having the bundle specifying the security details requested, while the system operator remains fully in charge. OSGi claims to in all likeliness provide one of the most secure usable application environments short of hardware protected such[**?**]. Needless to say, security is very important when working with a system where lethal accidents are as prone as with roadside traffic.

2.3 Reactive Blocks

Reactive Blocks is a visual tool for building robust and flexible Java applications [**?**]. Boiled down, Reactive Blocks is a Java code generator with a graphical interface comparable to SDL. As opposed to traditional coding, Reactive Blocks has a visual representation that focuses to a larger extent on the flow of a program, for making event-driven and concurrent systems. The only requirement for running a Reactive Blocks application is the capability of running Java. This means it can run applications on a wide array of hardware, though it may need some more resources than what is available on small embedded systems.

2.3.1 The Reactive Blocks Architecture

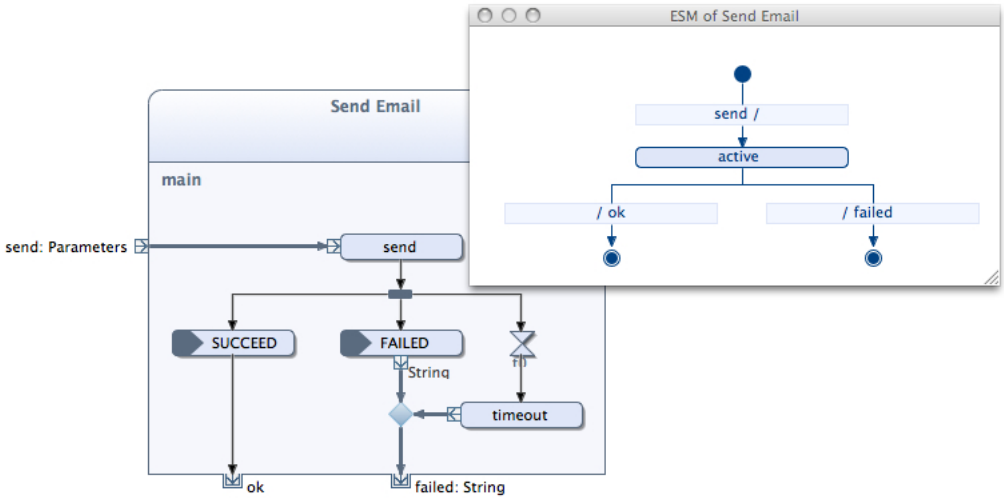
Systems, and subsystems, in Reactive Blocks are designed using reactive building blocks and connecting these to each other. A reactive building block consists of three parts described as follows [**?**]:

– An activity diagram that describes the internal behaviour of a block, the internal flow and logic. In Figure **??** we can see the activity diagram of a block called Send Email. In this case, the block starts with the input "send" which triggers the operation "send". After "send" has started the block either gets a "SUCCEED" event, a "FAILED" event or goes through a timout timer. If the "SUCCEED" event happens the block outputs ok, and if "FAILED" or the timeout timer runs out(and triggers the timeout method) the block outputs a string "failed".

– Java methods that describes the logic of operations.

– An external state machine (ESM) which works as an interface to the rest of the system, describing the legal sequence in which its features can be used. In Figure **??** an ESM is shown. The ESM shows what inputs can be received and outputs can be sent in which states. In the case of the Send Email block shown in the figure, in the initial state it can receive a "send" signal through the "send" pin, which transitions it to the state "active". From state "active" either output "ok" or "failed" are possible, both of which causes the transition to the final state of the block.

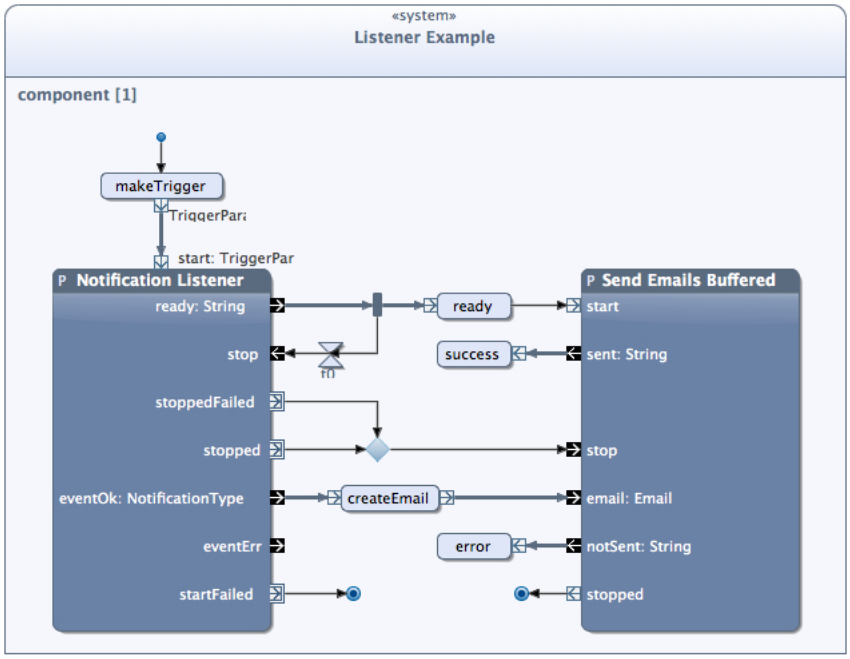
Figure 2.5: A generic reactive building block in Reactive Blocks (a local block). The Top-right view shows the ESM of the block, and the bottom-left view shows the activity diagram. Image from http://reference.bitreactive.com/reference/types-of-blocks.html



In addition to the normal parts of an activity diagram a reactive building block can contain reactive building blocks. There can therefore be multiple layers of reactive blocks in a single system. On the topmost level the block is called a system block, and it differs slightly from other blocks in that it does not allow for input or output parameters, instead it has initial nodes and activity final nodes that marks the start and termination of the system. In Figure **??** we can see an example of a system block. In the example block the initial node is the one in the top-left corner leading to the makeTrigger-operation, and the activity final nodes are the ones that follows the startFailed output and the stopped output. Notification Listener and

Send Emails Buffered are both local blocks. Every application needs a system block to be complete.

Figure 2.6: An example of a system block in Reactive Blocks. Image from http://reference.bitreactive.com/reference/types-of-blocks.html



2.3.2 Benefits of Reactive Blocks

There are a lot of benefits to using Reactive Blocks for system development. Some of the strongest ones are as follows.

Concurrent The inherent concurrent and event driven design of Reactive Blocks systems makes designing concurrent systems a lot less complex than it does with pure code. A lot of development problems come from the complexity of handling concurrent behaviour without utilizing the power of state machines. This is simplified with Reactive Blocks, with its built-in runtime system and modelling being part of the implementation instead of a precursor to it.

Reuse In a similar manner to that of the OSGi platform, Reactive Blocks is built from small dividable parts, though instead of modules, there are blocks. These blocks are very suitable for reuse. The reason why Reactive Blocks are more reusable than the norm, as normal Java is also to some extent reusable, is because of their

improved interfaces. The problem with traditional reuse is that there isn't really a good way to inform systems that wants to reuse code what the flow and thread compatibility is through APIs. In Reactive Blocks on the other hand, these are well defined for blocks so there can be reuse with no fear of deadlocks even if the actual implementation encapsulated by a block is unknown[**?**].

Figure 2.7: Reactive Blocks provide more information through their interfaces than traditional APIs. This ensures that they work together very well, and have a good disposition for reuse. Image from http://reference.bitreactive.com/papers/secret-twists.html

	Traditional APIs	Reactive Blocks
Method names	✔	✔
Types	✔	✔
Sequence of calls	-	✔
Execution times	-	✔
Thread compatibility	-	✔
Truly bi-directional	-	✔

Visually represented Since modelling is an intrinsic part of producing code with Reactive Blocks, there will always be visual representation of the applications that are being made, that describe the actual system. In application development where the modelling is not part of the implementation it self, there is no guarantee that it actually represents the system it is meant to represent. In Reactive Blocks this is not the case. The benefits of this are significant, the flow of the system can be shown in a clean manner, shown in Figure **??**, and structuring the code is much easier. Instead of ending up with a giant complicated code base you have a hierarchy of building blocks.

Figure 2.8: The flow of a program is illustrated, where the left side shows code, and the right side shows graphical building blocks. The dotted arrows would not be visible in most editors. Image from http://reference.bitreactive.com/papers/secret-twists.html

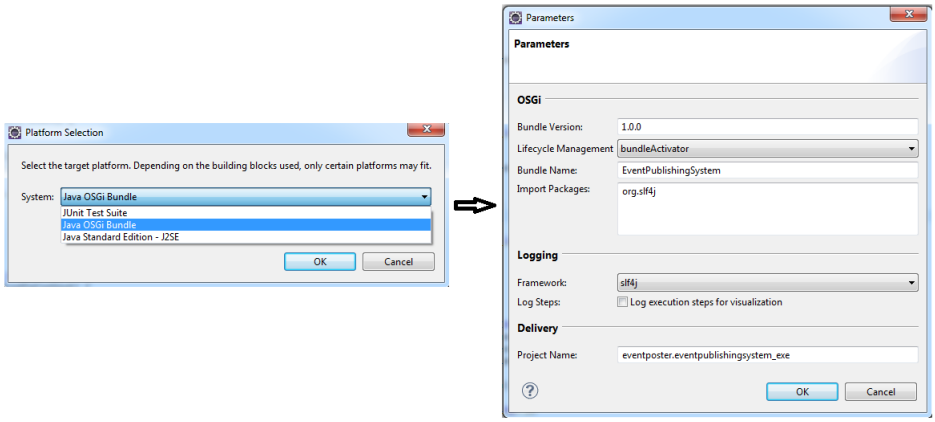
Verification Reactive Blocks has a built-in verification tool that does a formal analysis of blocks and systems uncovering problems such as starvation, deadlocks and race conditions that are not revealed with traditional testing. The information contained by the visual design of the blocks, and their behavioural contracts is what makes this possible. In Reactive Blocks an analysis of all possible scenarios an application goes through is done with the click of a button, and in most all cases it takes less than a second to perform[**?**].

Figure 2.9: The different type of errors in a system are found in different ways. Most software technologies utilize all but the formal analysis, which means that some errors might go unnoticed. Image from http://reference.bitreactive.com/papers/secret-twists.html

2.3.3 OSGi-Reactive Blocks Integration

Reactive Blocks comes with built in OSGi integration. As seen earlier in the chapter, both OSGi and Reactive Blocks bring a set not exclusively overlapping benefits to the table. Fortunately there is no need to pick one of the two if one were to make an application. Since Reactive Blocks is not a programming language in itself, but a code generator with a graphical UI, it could, in theory generate code for any platform. As of now, Reactive Blocks has support for both basic JAVA, and OSGi. That means that with just a choice in the build options, see Figure **??**, of Reactive Blocks the application can be built as an OSGi bundle, or a Java project. This means we can build applications with both the benefits of OSGi and Reactive Blocks with very little extra cost compared to Reactive Blocks that builds to vanilla Java.

Figure 2.10: To build OSGi bundles in Reactive blocks the only requirement is to chose OSGi from the platform selections. If needed, any extra parameters for the manifest can be added in the parameters menu. Image from a generic Reactive Blocks project.



2.4 Chapter Summary

In this chapter the technologies that are being used in the thesis were explored. The technological field of ITS, which is what the specific technologies of Reactive Blocks and OSGi will be tested towards. ITS is a traffic system that incorporates technology to add intelligence. This entails data gathering and communications as well ass more well informed decision based signalling. The specific technologies explored in the chapter are OSGi and Reactive Blocks. They are used together, where development works mostly in Reactive Blocks, but the code generated builds to OSGi. OSGi is a component system for Java that utilizes jar-files to create modules. The goal is to have a scalable platform to build dynamic systems. Reactive Blocks is a visual tool that builds code from blocks made of activity diagrams, state machines and Java methods. Reactive Blocks utilizes architecture modelling that has a direct relationship to the generated code, instead of an ideal architecture modelling that may only exist on paper. Both OSGi and Reactive Blocks have their benefits, and they are well integrated, so using both together is almost the same as using Reactive Blocks that generates plain Java.

Chapter 3
Current ITS at Statens Vegvesen

Statens Vegvesen is allready in the process of developing their ITS in Norway. They are working with Volvo and their ITS-project to make the roads safer and more effective by making them smarter. The main testing site for Norway's ITS-system is located in Trondheim, and the project is currently in a Prototyping stage of development. In this chapter the status quo of the ITS project at Statens Vegvesen is explored, as well as why it is so.

3.1 Technology Specifications

Statens Vegvesen has made a specification for the roadside ITS that outlines the requirements that must be fulfilled. In this section those specifications are summarized roughly, with the goal of finding whether they are compatible with the main technologies of the thesis.[**?**]

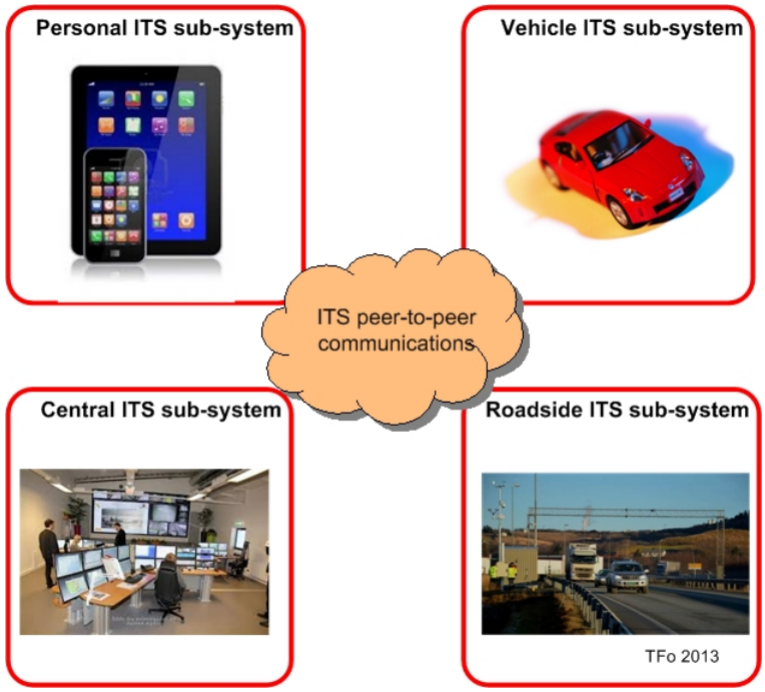
3.1.1 System Architecture

The ITS architecture is divided into four subsystems, shown in Figure **??**, they are described as follows:

- The Roadside ITS sub-system, which consists of anything that's on the side of the road, including sensors, signs and ITS-stations.
- The central ITS sub-system, which is the brains of the operation, it is where the data storage is located.
- The vehicle ITS sub-system, which is exactly what one would think, it is the part of ITS that is in vehicles. This provides a vehicle-specific ITS-station that communicates with the roadside ITS.

– The personal ITS sub-system, which is any ITS applications running in hand-held devices of the everyday traffic user. This will be one of the ways to bring the data from the ITS to the public.

Figure 3.1: The 4 ITS subsystems in the architecture of Statens Vegvesen. Image from [**?**]



The observant reader might have noticed that two of these are much more under the control of the product owner of ITS on a national scale than the other two; the central and the roadside sub-systems. Statens Vegvesen has little control over the technology that's included in cars. Though the implications of supporting some technology might make some vehicles superior to others by virtue of communications with ITS-stations, the technology choices in the vehicles is mainly up to the manufacturer. Fortunately there are some large projects working on a standardized interface and communication protocol for vehicles in ITS, some of these will be explored further in **??**, and **??**. This means that apart from communicating with vehicles the ITS project at Statens Vegvesen really isn't concerned with the structure on the inside of the vehicle sub-system. This is also, but to a smaller degree true with the personal sub-system, as this is not planned yet, and may well be open to the public in contrast

to the roadside and the central sub-systems. This means that the implementation and building job for the ITS project at Statens Vegvesen lies almost exclusively in the central and roadside sub-systems. These will be explored further in sections **??** and **??**.

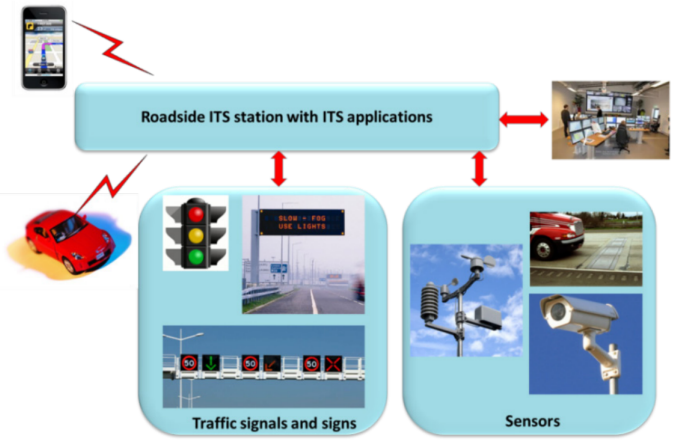
3.1.2 Central ITS sub-system

The central ITS sub-system handles the heavy loads. This is where the data storage is, and it is also where any heavy duty calculations would be performed. The data gathered at the roadside sub-system will be passed here, and any functionality that relies on heavy data loads will be here.

3.1.3 Roadside ITS sub-system

The roadside ITS has direct communication with all the rest of the sub-systems, and can work as a middleman for all cross-sub-system communications. The roadside ITS station is the brains of the roadside subsystem, it controls signals and signs, and gathers data from sensors. In addition to this it communicates with vehicles, personal systems and the central ITS. The roadside ITS subsystem is the main data-miner of the ITS, and it provides communication between the entire system. In the prototyping part of the thesis, the prototype will be made for the roadside ITS station.

Figure 3.2: The roadside ITS subsystems in the architecture of Statens Vegvesen. The zig-zag lines denote wireless connections, the arrows denote continuously open interfaces that can be either wired or wireless. Image from [**?**]



In the rest of the section the focus will be on the Roadside sub-system, as that is where the focus of this thesis lies.

3.1.4 Functional Requirements of the Roadside ITS-stations

The first step to finding out whether some technology is a good, or even a viable, choice for a system is to see whether it can cover the functional requirements of said system. The functional requirements of the roadside ITS-stations are as follows [**?**]

- Road network management; manage road network information and quality.
- Utilisation management; monitor the traffic situation, perform traffic control, provide traffic situation information.
- Vehicle Management: monitor vehicle and driver behaviour, support vehicle operation and emergency management, and manage vehicle information.
- Provide information services: provide traffic situation-, road network-, travel-, environmental-, and tourist information services.

These requirements effectively means that the roadside ITS-stations are responsible for data collection and distribution, as well as controlling signalling systems, with some extra detail as to what has to be covered in these areas. These requirements will be considered further in relation to Reactive Blocks and OSGi in **??**.

3.1.5 Security Requirements of the Roadside ITS-stations

The integrity of the information gathered and used by any system is important. When the system pertains to environments where errors can lead to fatalities, the importance of trustworthiness and reliability does not lessen. Therefore it is required that ITS-stations shall protect information collected, handled and stored from unauthorised access(confidentiality), protect information from unauthorised changes or deletion (integrity), and provide the required information needed for processing ITS applications(availability) [**?**].

3.2 Vehicle-Roadside interface

Since the system that's delivered by Statens Vegvesen only runs on hardware on the roadside, central, and possibly personal ITS sub-systems there must be some common interface to manage communications with vehicles. In this section one such interface will be explored, in order to understand the context in which the sub-systems operate. Since the functional requirements of Roadside ITS-stations [**?**]

include communications with vehicles, the interface between them is also an inherent member of both sub-systems.

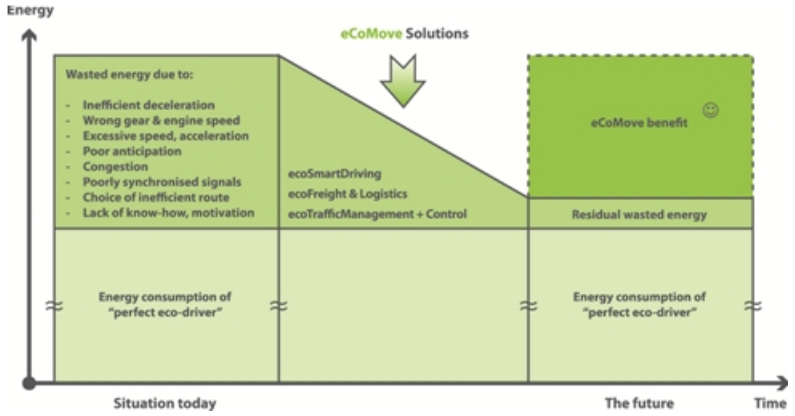
3.2.1 Cooperative Awareness Service

An integral part of ITS is communication between entities in the traffic system. One of the ways to make that happen is through the Cooperative Awareness service. CAS is an ETSI standard service that is used to share data on the roads, between entities like ITS-stations and vehicles. The messages used in CAS are called Cooperative Awareness Messages or CAMs. CAMs are sent over the 802.11p protocol, wifi for vehicles. The goal of this service is to give further knowledge and information of the situation on the road to, initially, the roadside stations. This in turn makes for a better understanding of the roads and ideally gives the means for more secure and better traffic flow. CAMs can contain information on the position of a vehicle and basic status. CAS is part of the Access Technologies of the ITS architecture, and it is the access technology that will be used for the purposes of this thesis. [**?**]

3.2.2 The eCoMove Project

The eCoMove project is a European ITS initiative focused on the use of ITS for energy efficiency. In contrast to other ITS projects, the focus of eCoMove is not with safety concerns. The most important part of the eCoMove project with respect to this thesis is their specialized CAMs called ecoMessages [**?**]. The reason we mention the eCoMove project is because with it comes some implications both to the possible benefits of ITS in general, and some functionalities that will be part of a more final ITS. The eCoMove project's additions to CAS will have to be integrated in any system that utilizes eCoMove. Ecomove is working towards standardisation, and there is good reason to assume that it will be implemented in vehicles in the not too distant future. Thus support for eCoMove is, if not vital, a good thing to either have implemented in ITS, or to have a structure that is compatible with.

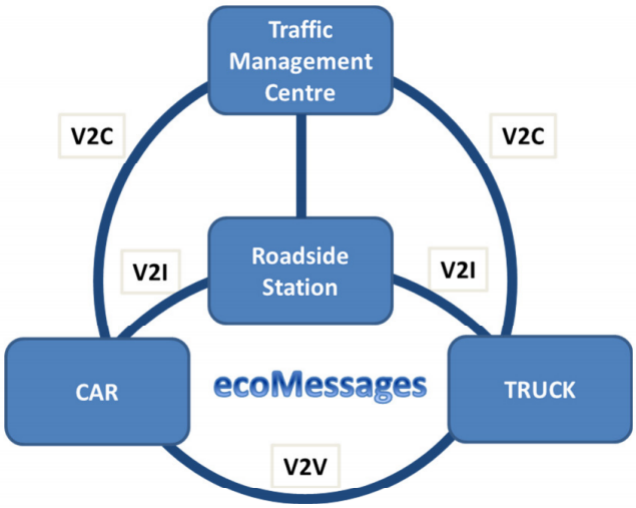
Figure 3.3: The eCoMove Project Vision. By applying optimisation of the traffic with regards to fuel-use, the eCoMove project aims to reduce the carbon footprint of the transportation sector drastically. Using data mining and path-optimization, etc. the drivers will receive help to drive more eco-friendly. Image from [**?**]



3.2.3 ecoMessages

ecoMessages are the packets, or messages, that replace CAMs when operating with the eCoMove project. They all share a common header, and are based on CAMs and the Distributed Environmental Notification Messages (DENM), both of which are ETSI-standards [**?**]. ecoMessages focus on location, intersection topology and speed advice. If an ITS should integrate the eCoMove project it has to support ecoMessages.

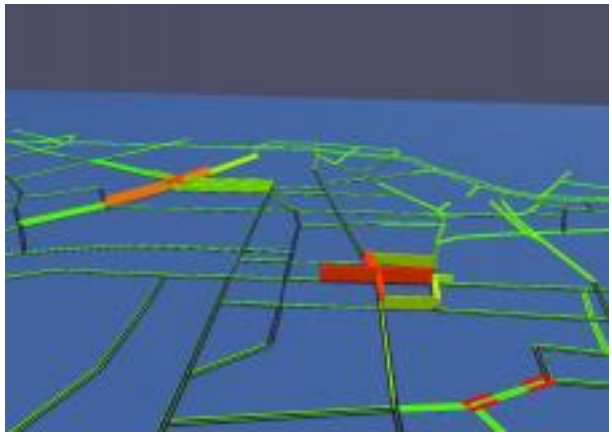
Figure 3.4: ecoMessages in the ITS are passed on all the channels that CAM's are passed on. They are can be used on all communications where at least one of the ends are a vehicle. Image from [**?**]



3.2.4 eCoMap

The eCoMap is one of the functionalities that eCoMove has been working on. It is a weighted graph where the weights are fuel consumption based on data gathered by roadside ITS-stations, and the edges are roads. This is one of the most important eco-friendly oriented functionalities in the eCoMove project, and serves as an indication to what can be done with big data in ITS.[**?**]

Figure 3.5: The eCoMap illustrated graphically, the weight is illustrated by color shade. Image from [**?**]



3.3 Example Application: ecoMessage Logger

For the prototype in this project an application running in the ITS system already will be remade with Reactive Blocks, to compare it to the current system and evaluate the benefits of using this alternative strategy.

3.3.1 ecoMessage Logger Description

The ecoMessage Logger logs incoming ecoMessages. The platform of the logger is ITS stations on the roadside, though it is conceivable that it could run in vehicles as well. The logger has the capabilities to send the received ecoMessages out to an external server, though it relies on services within the OSGi bundle context for this.

3.3.2 Analysis of ecoMessage Logger

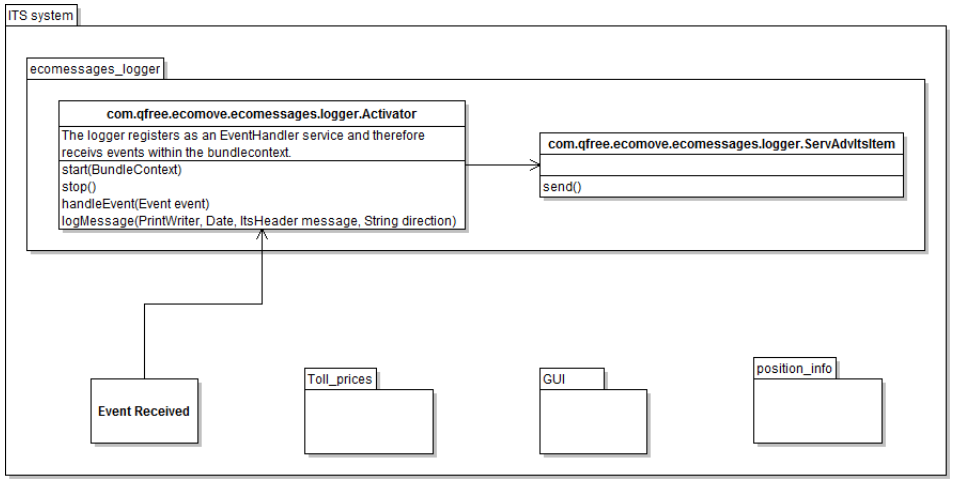
The Logger consists of two relatively big classes, with a few methods. The majority of the code in the classes are in a minority of the methods, with the longest method being more than a seventh of the application.

Table 3.1: Data from the ecoMessage Logger Classes

	Activator	**ServAdvItsItem**	**Total**
Lines of code	592	178	770
Methods	17	14	31
Average Method Lines	31	9	21
Longest Method	120	67	120

The ecoMessage Logger is integrated in the ITS station system as a bundle that advertises itself as an EventHandler Service, meaning that any events that should be handled by an event handler within the bundle context will be passed to the logger. They may be passed to other event handlers as well. Upon receiving events the logger checks whether the event is one that it should handle, and does so if it is. If it receives an incoming ITS event it will start the process of logging it Figure **??**.

Figure 3.6: The ecoMessage Logger in the Vegvesen ITS[**?**]



ecoMessage Logger Module Summary The ecoMessage Logger is a quite simple application that illustrates the communication between modules in OSGi. It does not know much of its surroundings, but relies on some other module handling incoming ecoMessages and using the eventAdmin to publish these to any modules that have shown an interest in these. In this case the module does not do much with the data apart from exclaiming to any actor listening for test purposes. Though the ecoMessage Logger does not have any important functionality beyond testing, it could easily be feeding the messages either after parsing or before to some server via any number of protocols. By logging messages it shows that the inner OSGi communication works as it should, at least in this case. The ecoMessage Logger will be expanded upon in **??**

3.4 Interviews with Statens Vegvesen

This section aims to give an overview of the problems and workings of ITS from the perspective of the people working on the Statens Vegvesen ITS Project in Trondheim. To get a better introduction to the actual routines and inner workings at Statens Vegvesen, a handful of interviews were conducted. The subjects of the interviews included developers, and product owners at Statens Vegvesen, as well as a researcher working with Statens Vegvesen at SINTEF. All information following in this section is based on the answers given in interviews.

The Greatest Challenges in ITS

In essence an ITS is doing two things. It is collecting data, and using that data in some way. One of these must be done in order to be able to do anything of the other.

That is collecting data. This seems to be the focus, and also the largest challenge when it comes to ITS at Statens Vegvesen's side of the operations. In the first place Statens Vegvesens task is to collect data, and make that data available. With this comes the challenges of handling big data, and of getting data that is good enough to be reliable, and usable. This is the first step and may be the greatest ITS challenge for Statens Vegvesen.

For the best use of the data, once data collection is there, the hope is that research institutes such as universities and SINTEF will take part in the work. Statens Vegvesen's part in this would be funding research projects, and implementing applications based on the results.

The Largest Costs in Transitioning to an ITS

On Statens Vegvesen side of things the largest cost seems to be perceived as lying in the development side. The impression is that once the system is running, it will be robust enough not to need as much upkeep as it needed development, and that the planning of it would not measure up to the cost of implementation. On the cost of development being much higher than the planning the product owner made an insightful commented on how this may be due to being used to too optimistic planning phases.

3.4.1 The Requirements of an ITS

The ability to provide data that is trustworthy and valuable is the top priority for most of the interviewees. The focus of Statens Vegvesen's ITS-project is data, which was made even more evident from the answers to what requirements are needed for an ITS. What this entails is both a secure channel, making the data trustworthy, as well as having the robustness to support almost non-existent downtime for continuous real-time data gathering. In addition to this, the product owner especially was interested in an easily expandable and patchable solution that would not suffer from legacy problems and extreme complexity once the first iteration was done.

3.5 Chapter Summary

The ITS project at Statens Vegvesen is currently in a planning phase. They are working with the industries to get to the point where they can provide the roadside equipment capable of communicating with vehicles over a standard protocol. At the moment of writing work is done on standardizing both with Volvo and with the eCoMove project. They are also working with SINTEF where OSGi is already in use for prototyping. The focus for ITS now, as seen from Statens Vegvesen is to get to a point where data is continuously gathered, stored, and made available. This

way they can apply the results being found by transportation research projects. The future holds roadside-vehicle communication and data storing, as well as funding and cooperation with research organizations.

Chapter 4
Challenges in ITS

In this section the main challenges, bottlenecks and most important requirements for an ITS are explored. To find the right way to solve a problem, one of the first things to do should be to identify and understand the problem as well as possible. In this case the problem is what is the best platform on which to build an ITS. To find an answer to this, there are two parts of the problem that we need to familiarize ourselves with, namely the nature of an ITS and the technologies under evaluation. This chapter is dedicated to the former.

4.1 ITS Requirements

To evaluate the challenges of a task, one must first explore what is needed to complete that task. In this section the requirements for a complete ITS, and their importance are explored. The section is divided into areas of responsibility such as functionality and robustness. This is to isolate as small requirements as possible, so that they can be easily identified and expanded upon in section **??**

4.1.1 Functionality

There are two groups of functionality in an ITS, namely data gathering and traffic control/guiding. As mentioned in chapter **??** there can be no informed controlling functionality without data. Even traffic lights in the traffic system currently in use uses data from button presses or sensors to know if pedestrians or vehicles are waiting to cross. There can however be data collection without traffic control applications. Because of this the first, and arguably the most important part of an ITS is the gathering and storing of structured data. Data gathering in ITS can be done in many different ways. One that has been mentioned earlier is through vehicle-to-roadside communication, but in addition to this there are lots of valuable data that can be gathered from sensors in roadside ITS sub-system. In the end we are left with the ability to collect data as the most important functionality requirement for ITS. In sum the functionality requirements are:

- Reliable gathering of real-time structured data.
- Controlling and informing traffic by use of the data.

4.1.2 Robustness

Though robustness is very important in an ITS, there are different levels to which it is important in different parts of the system. It can for example be detrimental, and possibly dangerous if the control for a traffic light, or a road barrier stops functioning. There are already some handling for this without the extra data and intelligence soon to be available, a traffic light will for example typically just blink orange if something is wrong, and the traffic should treat it as if the crossing had no signals. For calculating the most energy efficient paths, for example, the uptime might not be as instrumental. Most of the time, the data used for this does not change very quickly, at least not in a way that makes large differences. So applications that gather data for this may not need to be as robust as the control applications. There still needs to be a good amount of robustness, even in the applications that are not the most time-sensitive. In sum the robustness requirements are:

- Robust enough applications to run continuously enough to support reliable real-time data.
- Robust and error-safe control functions to avoid creating dangerous situations.

4.1.3 Security

Security is important when the integrity of data is important. There are different aspects to the security in systems such as an ITS, and they serve different purposes. On one side security is there to make sure all the data that's collected is right, and that it comes from where it is supposed to come from, and on the other side it is making sure that the data does not end up in the wrong hands. The last part is really only an issue when there is personal data involved, at least in the ITS at Statens Vegvesen. This is because most of the data will be open to anyone and everyone anyway, and is therefore not sensitive. What is important for all the data however, is that it is trustworthy. If it is not, then it is useless, and does not guarantee a good representation of the reality. Additionally all control systems must be secure. If control systems in Norway were to be hacked, it would not be the first time something like that had happened [**?**]. The security requirements can be boiled down to the basic principles of information security[**?**]:

- Integrity. For the data that is used and gathered to guarantee accuracy and correctness, it must not have been tampered with anywhere in its life cycle.

It must therefore be protected from modification by unauthorized actors. For integrity we therefore need to fulfil:

- Authenticity. It is important that the sources of data are what they say they are, in order to be sure that the data reflects the reality. That means that authentication is needed.
- Non-repudiation. The origin and endpoints of actions on the system must be non-reputable in that after it is done, the actor can not deny having acted and the receiver can not deny having received.

– Availability. Since the data is to be used in real time, and be available at all times, the system providing the data must be safe from denial-of-service-attacks that could shut it down.

– Confidentiality. When there is personal data involved, which may well be the case when communicating with vehicles, the data privacy must be kept. If not because it could be dangerous if it came in to the wrong hands, then because confidentiality is often required by law when dealing with most personal data.

4.1.4 Light-weight

For some applications the requirement of being light-weight is very important, for most ITS applications this is probably not the case. The functionality that needs to run on a roadside station or in a vehicle is already pretty light-weight by nature. And with the way pocket sized computers have come in later years there is tremendous computing power available in small and cheap units. A raspberry pi for example is cheap and has the power to run a system that would have been considered heavy not too long ago[**?**]. Most heavy applications can be running in the central ITS sub-system, very sensibly. In short there is not much concern for making extremely light weight solutions, this off course does not mean that optimization is a waste of time.

4.1.5 Extensibility

An ITS is by its nature a system of high and dynamic granularity, and will therefore have to be changeable, and hospitable for added functionality. There will be no final implementation of the road network in Norway, or the technology that runs on it, in the foreseeable future. Any ITS must therefore have the ability to host new functionality without revamping the entire system for each add-on. This means an ITS needs to be patchable, and extensible. In short the extensibility requirements are as follows:

- Adding on extra functionality must be as trivial as possible, to a reasonable degree.
- Patching old functionality should be as simple as possible, and not cause issues when put into production

4.2 ITS Challenges

In this section the challenges of making an ITS, or rather fulfilling the requirements of an ITS are explored. The section builds upon the points from the previous section, with focus on what barriers need to be broken to support the existing requirements. The section, as the rest of the thesis, focuses on software, not hardware challenges.

4.2.1 Functionality

The functionality challenges of an ITS are the challenges that must be overcome to fulfil the requirements. The first and foremost requirement, as discussed in **??**, is to provide gathering of real-time structured data.

Gathering data To make this happen, there are multiple steps that need be taken: Setting up the infrastructure needed, namely road-side stations, sensors, and a central data station. And building applications that communicate over the infrastructure. Quite possibly the largest challenge here is to get vehicle ITS-stations that work interchangeably in most-to-all cars. As car manufacturers are somewhat individually responsible for what a car is equipped with, projects with the goal to standardize such equipment are probably the best approach to solve this problem. As for producing roadside ITS-stations there is a whole other thesis to be made for which choices are best suited, be they cheap single-board computers or larger more expensive and more powerful computers. Once the infrastructure is there, the gathering of data should be quite possible to achieve with software systems.

Control applications To be able to make control applications is not much different from being able to collect data. There just needs to be some output device to use as output, and somewhere to run an application. Some such applications will most likely run on the same stations that gather data, only with a different output.

4.2.2 Robustness

To meet the robustness requirements there are two things that must be supported, up-time, and error-handling and -avoidance. To maintain up-time there are a lot of factors that needs to be taken into account, such as hardware-failure, denial-of-service attacks(see **??**), and patching related down-time. To some extent all of these must be handled by the software system, and will have to be solved.

4.2.3 Security

For the security requirements to be fulfilled there are tons of research done in the sciences as to what would be the best solutions, in this section they will be run through on a very general and shallow manner. To reach the goals of the core principles, section **??**, the following steps must be taken:

- Integrity. To maintain integrity there must be some form of authentication, that maintains non-repudiation.
- Availability. As with any data center that is to be open to the public it needs to be protected from DoS-attacks or other malicious actions that can bring it down.
- Confidentiality. Sensitive data has to be encrypted, and protected. This involves authorization as well as authentication.

4.2.4 Light Weight

Seeing as the applications don't really need to be running on sparse resources making them as light weight as possible is more of an optimization point than a basic requirement. There is therefore not a large barrier to pass in the light-weight area of ITS in the time of writing.

4.2.5 Extensibility

Making applications that don't get more and more complex the more it is worked with is very difficult, and has cost countless man-hours for projects that have not been doing a good enough job of it. There are a few ways to go about getting an understandable and workable code-base. One is to have very strict clean code standards and practices [**?**]. Another is to work with modular programming, so that to change one part of the system there is no need to know the rest of it. An even better way to achieve extensibility is to follow both the aforementioned practices.

4.3 Chapter Summary

In this chapter the challenges and requirements of an ITS have been explored. In short, there are two main challenges, gathering data, and finding the best way to use said data. The first part, gathering data is not an abstract task, and can be started and to some extent finished in the near future. As for the best ways to use the data that will be available, the best approach may be to fund and create an environment for research projects in the field of traffic research.

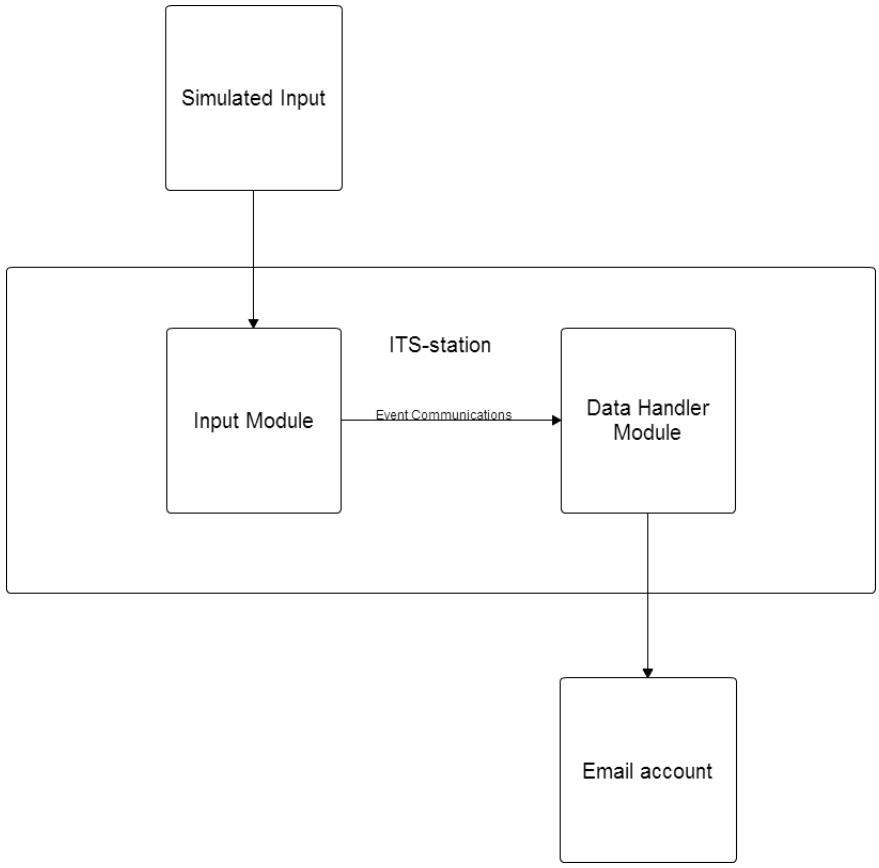
Chapter 5
Building ITS-blocks with Reactive Blocks and OSGi

In this chapter the process of making a module for ITS are explored. The goal of the chapter is to find out whether it is possible, conceivable, and practical to use Reactive Blocks and OSGi for ITS. This will be based on the prototype module made with Reactive Blocks and OSGi. The results of the chapter will be used in the discussion chapter to further explore the viability of Reactive Blocks and OSGi in ITS.

5.1 Prototype

In this section two small modules that could run on a completed roadside ITS station that has been made are explained. The modules are a logger that could replace the one described in **??**, and an input simulator that simulates input to the station from the outside environment. The goal of the prototype is to highlight some of the strengths of both Reactive Blocks and OSGi, and how modules interact in a scalable system. The modules use the event communications in OSGi to cooperate. The input module simulates data from the outside environment, and shares that data with the rest of the system without knowing how the data will be handled after it is made available to the global context. The logger module logs data via emails once it is made available to the system. The Events are passed via the EventAdmin running in the OSGi context. First the EventAdmin will be explored, and then the modules. The code for the prototype can be found in appendix **??**.

Figure 5.1: The conceptual architecture of the prototype modules in the ITS station environment



5.1.1 communications

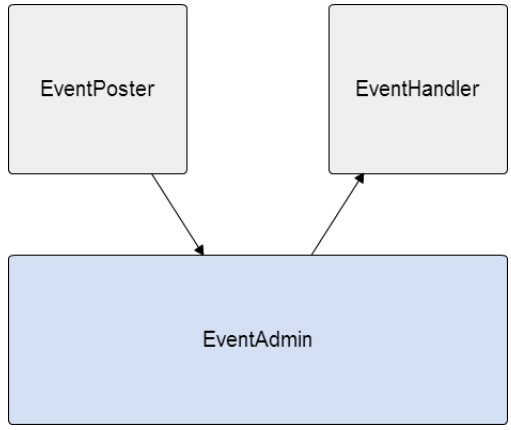
To make an OSGi application as dynamic and extensible as possible, there are some global ways of communicating between modules. This is most commonly done through the service register, described in section **??**. To make the communication in OSGi available for Reactive Blocks modules two blocks were built. Their description follows, as well as an introduction to the OSGi EventAdmin.

EventAdmin

To understand how and why the modules work, and work in a very integrable way, we must first understand the event-handling of global events in OSGi. The EventAdmin in OSGi is used to pass global events to the rest of the OSGi context, and it is one of the main communication buses in an OSGi system. This means that events that one module receives can be made available to the rest of the system

by use of the EventAdmin service. That sounds very much like the perfect way to distribute received messages from vehicles and sensors at a roadside ITS-station. This way any socket, or other listener, can be concerned only about listening to incoming messages and pass them to the rest of the system to be dealt with by some other module. This utilizes the power of modularity by dividing responsibilities and functionality in small modules that are easily replaceable or improvable even in a large system.

Figure 5.2: The event communications in OSGi, the arrows shows the flow of events



Event Publisher

For the communication model through the EventAdmin to work, there must be a publisher, and a receiver of events. In theory, there could be only one of the two, but in that application not much would be done. The event publisher is a fairly simple module that publishes events to the EventAdmin. It is simply a straight forward way to pass global events to the rest of the system. This way, any input module, such as a listener for CAMs or sensor data can worry only about listening, and nothing else.

The EventPublisher block, shown in Figure **??**, has two inputs. They are two different ways of publishing events to the system, the only difference is that one publishes asynchronously, and the other one does not. An alternative implementation, to simplify the use of the block, would be to only use one of the two ways of publishing events. The ESM of the EventPublisher block Figure **??** shows us that it goes from its initial state to its final state with one input, so a new block gets started for every published event. This means that the block has its own flow no matter if the publishing utilizes asynchronism or not, therefore having the option for both is unnecessary.

Figure 5.3: The EventPublisher Block

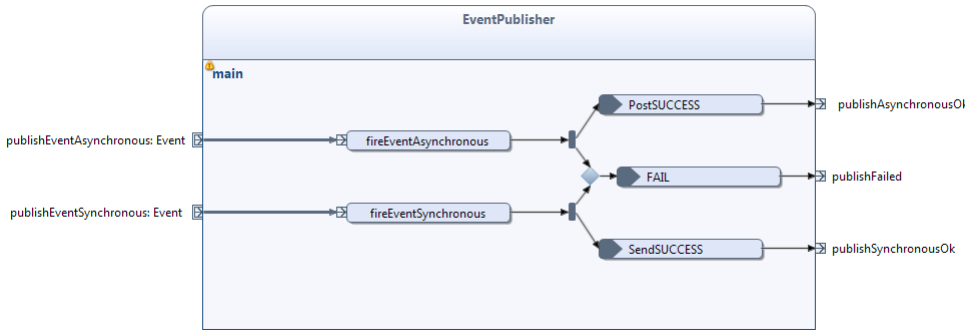
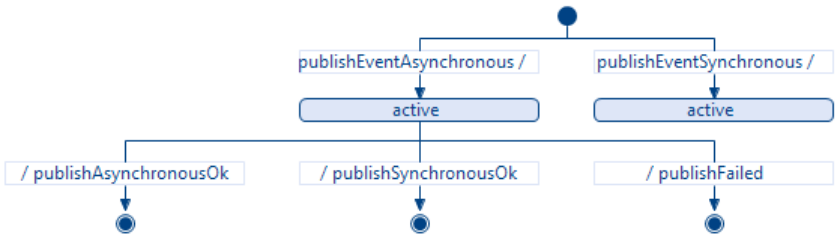


Figure 5.4: The external state machine of the EventPublisher Block



Event Listener

On the receiving end of the global event communications in OSGi are Event Listeners, or Event Handler Services. To make an as generic event listener as possible, a block that takes event topics as input was made, that subscribes to all global events of the topics. The Event Listener block is shown in Figure **??**. After being initialized the EventListener block outputs all events that match the topics given on initialization, until it is terminated. This behaviour follows the state transitions described in Figure **??**.

Figure 5.5: The EventListener Block

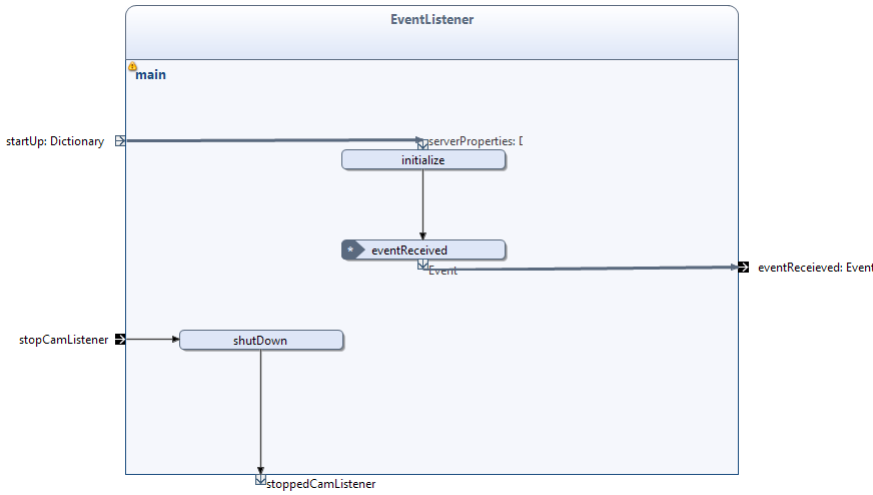
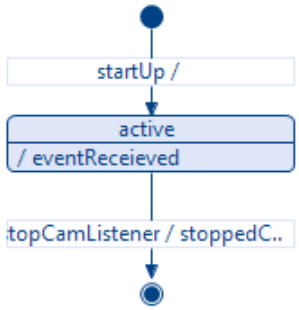


Figure 5.6: The external state machine of the EventListener Block



5.1.2 ITS prototype modules

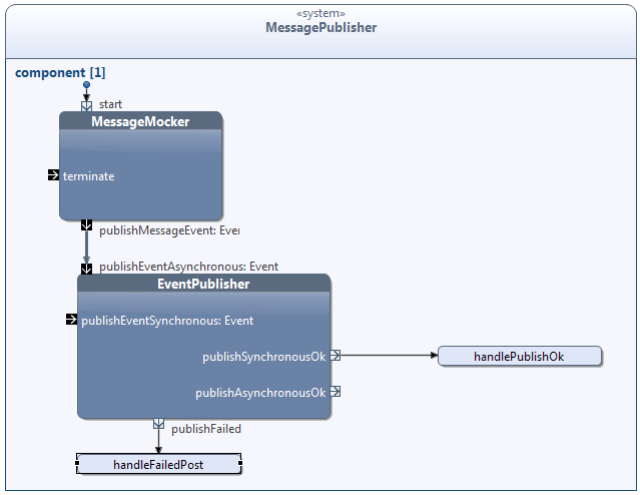
Now that the communications have been made available for Reactive Blocks modules, we can move on to an illustration of a real world system. In this system, there is one module that listens for incoming CAMs, and one that processes them once they have been passed to the rest of the bundle.

Reactive Blocks and OSGi message publisher module

The message publisher module, shown in Figure **??** is a module that publishes CAM and ecoMessage events to the OSGi system. In the case of the prototype, these messages don't come from an outside source, but are simply made in the

MessageMocker block and passed to the EventPublisher. Though this design does not receive input from outside sources, the same design would in all likelihood work well with for example a socket listener listening on a port reserved for 802.11p communications receiving CAMs from vehicles on the road.

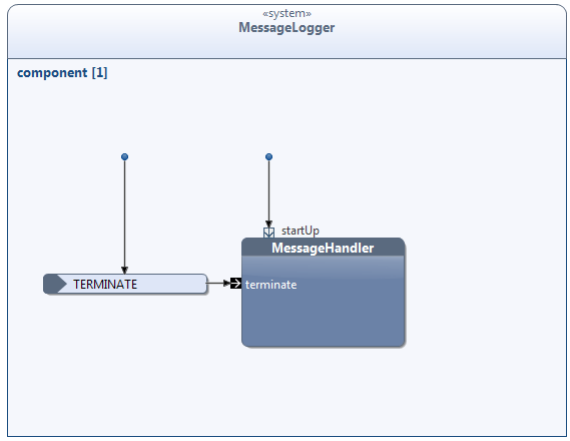
Figure 5.7: The system block of the message publisher module



Reactive Blocks and OSGi logger module

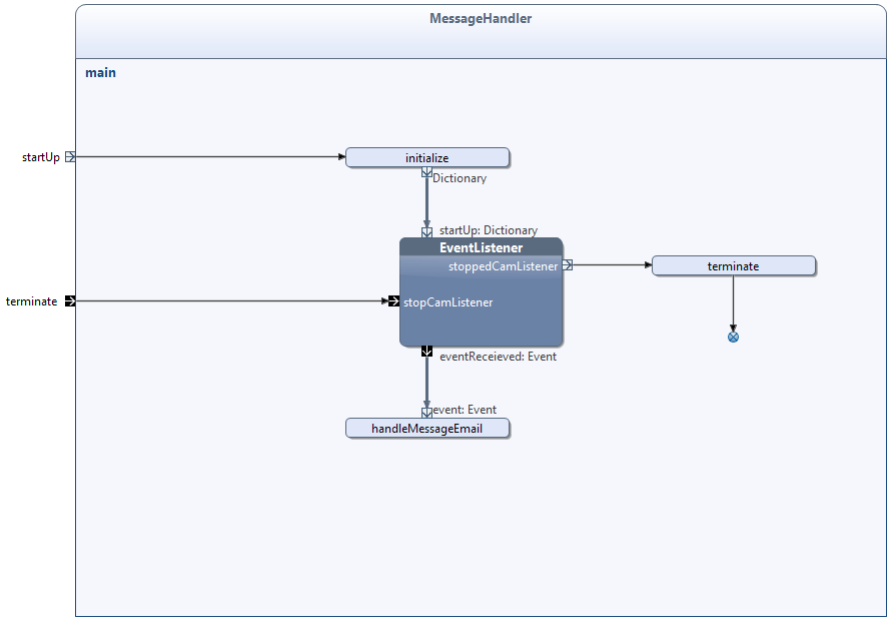
The Reactive Blocks logger module Figure **??** is a module that simply uses the EventListener **??** to catch any events with topics matching that of ecoMessages and CAMs. This module is the Reactive Blocks counterpart to the ecoMessage Logger analysed in **??**. Apart from using the Event class of OSGi, all its OSGi related activity is confined to the EventListener Block. The module consists of a system block with one local block, the message handler. There could well be many more, but since the functionality of the module is fairly simple this is all that is needed. The block is the MessageHandler block, shown in Figure **??**.

Figure 5.8: The system block of the MessageLogger module



The MessageHandler block, Figure **??**, Starts an EventListener with the topics of CAMs and ecoMessages as parameters, and runs the handleMessageEmail every time the EventListener outputs an event. Because of the lack of a central ITS sub-system available at the time of making the logger, it sends an email for every received message. The emails contain the message type, and the data attached to the OSGi event. The messageLogger continues handling message events until it is terminated.

Figure 5.9: The MessageHandler Block



5.1.3 Prototype Results

When running both the bundles in the prototype the output from the message logger module corresponded to all messages made in the message publisher module. Though this does not come as a surprise, it does show that using the EventAdmin as a communications bus in an application opens for a simple and extensible design. Any number of modules utilizing the incoming messages could be running side by side, and any number of modules listening for incoming messages from input devices could be running similarly.

5.2 Compatibility of Reactive Blocks and OSGi with ITS

The prototype shows the power of the modular structure of OSGi and Reactive Blocks, as well as the communication bus for Events via the EventAdmin. After building the framework of a very simple ITS-station with the prototypes, it is fairly straight forward to see how additional modules utilizing the same functionality through reuse of the communication blocks could be added with relative ease. This fits particularly well within a system where there could be any number of input and output devices available. There is no problem in using this same design for controlling traffic lights or speed sensor stations, the only difference is how the events are handled in the corresponding handler module. Using OSGi these modules can even be added in

runtime, with no down-time at all. The implications of this is that Reactive Blocks and OSGi are a very good match, at the very least with the roadside ITS sub-system.

5.2.1 Reuse

The prototype uses two quite generic blocks that can be used for event communications in any module that runs OSGi. The EventPublisher block and the EventListener block. If the event communications of OSGi is used for context-wide communications these can easily be reused in any new module. As shown by the modules of the prototype, they don't even need to be fully understood by the rest of the module in which they would be placed. With Reactive Blocks this reuse is as simple as a drag and drop and connecting the dots.

5.2.2 Extensibility

With the information being made available through system-wide communications such as the EventAdmin new functionality can be added at any point. The modules that were made in the prototype do not even need to know the implementation of the others, with the exception of knowing the common event topics. Additional modules adding to the same environment could be added without causing trouble. There is no added complexity to the system by adding new externally simple modules, even if the modules themselves were internally complex. As long as anything that does not need to be shared outside of the modules are not, the system as a whole does not increase more than it needs to in complexity when new modules are added, and can be extended as long as the host has the power needed to run it.

5.2.3 Customizability

Roadside ITS-stations may very well be subject to varying environments, with different kinds of sensor inputs and signal outputs. To make ITS-stations that can host new technologies at any time when they arrive and are installed, the software must be either customizable, or all-covering. The approach that is arguably best in the long term is to make customizable software. With the use of a design such as the one demonstrated in the prototype, modules that support different technologies can be added or removed when said technologies are made available or unavailable to the station. This makes an easily customizable solution that requires little work in order to accommodate a plethora of varying environments in individual ITS-stations.

5.3 Chapter Summary

In this chapter a simple prototype consisting of two modules utilizing the OSGi EventAdmin was explored. The prototype was built in Reactive Blocks and had one

endpoint where data was generated, and one that handled said data. The modules were running completely separately, and had no knowledge that the other module existed. That means they did not add to the complexity of each other, and that any number of other modules could be running and adding to the same functionality provided by the two that were made. Not only did the prototype not show any inclination that building an ITS-station with Reactive Blocks would be impossible, it showed that it would aid the process with significant benefits. From the findings in the chapter, Reactive Blocks and OSGi looks to be perfect fits with ITS, at the very least in the roadside ITS sub-system.

Chapter 6
Alternative Technologies

The previous parts of the thesis have been focused mainly on how well Reactive Blocks and OSGi works in the environment of ITS, but there are many other alternative technologies to choose from when designing an ITS. To find out if a technology is the right choice, it doesn't just need to be a good match, but it must work as well or better than other technologies that are also available. In this chapter the pros and cons of some of the most popular technologies that could be considered instead of OSGi and Reactive Blocks are discussed.

6.1 Java

Java is one of the most widely used object-oriented programming languages in the world [**?**]. It runs on the Java Virtual Machine (JVM), and is the basis for both Reactive Blocks and OSGi. Its first version was released in 1995, and the community around it has grown steadily since. In the latest version of Java as of 2014, Java 8, support for functional programming and improved security and parallelization has been added, making a better version of an already popular language.

6.1.1 Benefits of Java

Community Java has an enormous amount of 3rd party libraries available for use. In many cases of generic functionality, code has already been written, and there is no need to write it again. Through reuse of libraries the power of a community of millions of programmers is available. In addition to having lots of libraries that can be used in new applications, the large following of Java means that there are lots of experienced programmers out there, that can work on Java projects.

Performance Java is relatively fast. Historically though, it was not always considered as a high performance language. It was just with the introduction of just-in-time compilation [**?**] in 1998 that Java started providing good performance. In addition to the JIT support, other performance improvements has been done, mainly through

optimizations in the JVM. Java today performs well compared to other just-in-time compiled languages, and even rivals C++ (a compiled language) in some numeric benchmarks [**?**].

Tools The developing tools for Java are many and varying in character. From light-weight text editors to heavy-weight integrated development environments (IDE's), and continuous integration tools. Having lots of different tools, means developers can work in the environment they prefer, which could increase productivity.

Platform Ubiquitous Java runs on a virtual machine, the Java Virtual Machine (JVM), that is designed to run on any platform with the same Java code. Any machine that can run the JVM can run Java programs, and that is quite close to any machine. Because of the JVM, Java programs can run on any platform desirable.

6.1.2 Java summary

Java is a powerful object oriented programming language, that has high performance and a large global following. There are a vast amount of resources to aid in Java development projects, as well as many developers with great knowledge of the language. A lot of projects choose to develop their systems in Java, and with good reason.

6.2 Scala

Scala is a functional object-oriented programming language. It is statically typed and meant for making components and component systems. It focuses on scalability, and the name is derived from "Scalable Language", it is a fitting language for everything from one-line expressions to large projects. It is chosen for use by companies such as Twitter, LinkedIn and Intel.

6.2.1 Benefits of Scala

Scala is a language designed with the intention to fix the shortcomings of Java, so as to make the building of component systems easier [**?**]. This is done both by joining functional programming and object-oriented structure, and by using statical typing. This makes Scala shine in some areas, some of these are described in this subsection.

Scalable Scala is built to be scalable. Its mechanics concentrate on abstraction, composition and decomposition, and not on having many primitives available for some level of the scaling. It is also a lot closer to an architecture that is a component system, which is a huge help to scalability.

Functional Scala is functional, and has all the benefits that any functional language has. Collection operations, for example are made a lot simpler, and quicker to code. Less noise in your code base will also, in all probability, make it more readable.

6.2.2 Scala summary

Scala is a language that has been built as an improvement on Java, without having to maintain backwards compatibility. It could therefore learn from the mistakes of Java and improve upon it, both adding new functionality, and trimming the fat that was not useful from it. It is a functional object-oriented language that is quite popular, and has a strong following. Even though Java now has some support for functional programming, it is much more likely that your Scala developers will know how to utilize that power, than your Java developers since it is so new. To sum it up, Scala is a viable choice for any project, with its strengths and weaknesses.

6.3 Akka

Akka is an open-source platform to aid in the making of concurrent and scalable applications [**?**]. It works with both Java and Scala, and utilizes the JVM for its applications. Akka is a relatively new technology, with its first official release in 2010, though it has already created an impressive following. Akka has been used by corporations such as Walmart, and LinkedIn [**?**] and is by its own testimony a good fit with automobile and traffic systems[**?**]. Akka has even been used for ITS sub-systems, in the Netherlands, with good results [**?**].

6.3.1 Benefits of Akka

Akka is a new technology that shows many similarities to both OSGi and Reactive Blocks. It is a platform that focuses on a reactive model, that is scalable and simplifies concurrent operations. It has many benefits, some of which are explored in the following paragraphs.

Scalable Akka is built to be scalable, and it is scalable. The use of an actor model, where the actors are extremely light-weight compared to normal processes, and cluster support makes scalability much better than that of many other technologies. The way fault-tolerance is handled in Akka also helps towards making scalable systems.

Fault tolerant Akka uses the "Let it crash" model for fault tolerance. Instead of focusing on fault avoidance, Akka assumes that faults happen anyway, and treat them as a natural state of the system. By linking actors to each other Akka manages to monitor the state of actors, to see if they are alive or not, and handle it if they

die. This makes Akka systems very fault tolerant, and deals with concurrent and distributed systems in a closer to real-world way than for example Java [**?**].

Tooling Like Java and Scala, Akka can be written both in powerful IDEs and light weight text editors. Though an IDE is often very helpful, it does not hurt to have the option of an approach even closer to the source, without all the heavy help provided from IDEs like Eclipse or Intellij Idea.

6.3.2 Akka summary

In summary Akka is a platform that has lots in common with the Reactive Blocks and OSGi duo, but differs in that it does not have the visualization aspect of Reactive Blocks, and it does not have the dependency control of OSGi. It does however support Scala, and does not need as specific tooling as Reactive Blocks or OSGi does.

6.4 Chapter Summary

In this chapter some development technologies, possible rivals to Reactive Blocks and OSGi, have been explored. Their respective benefits and structure have been reviewed, with the environment of ITS in mind. Two programming languages, Java and Scala, and a component system, Akka, were considered. Many other languages and tools are available for designing software systems, these may offer other benefits. The exploration of the platforms described in this chapter is meant to offer an insight to what else is out there. The findings in this chapter are used for the discussion in chapter **??**.

Chapter 7
Discussion

This chapter discusses the results found in chapter **??** with respect to meeting the challenges explored in chapter **??**. Reactive Blocks and OSGi is also measured against the technologies in chapter **??** in an attempt to answer the core question of the thesis: whether Reactive Blocks and OSGi is a good choice for developing ITS.

7.1 Meeting the challenges of ITS

The first and foremost priority when seeing if a technology is a good choice, is whether or not it is capable of meeting the challenges that the task requires it to meet. In this section the degree to which Reactive Blocks and OSGi can meet the challenges of ITS is explored, with the goal of finding out whether they are a viable option for making ITS, as well as a good one.

7.1.1 Functionality

The most important functionality that needs to be covered in an ITS, as discovered through interviews with the workers at Statens Vegvesen and the requirement specification of roadside ITS-stations, is to handle Data Gathering. This was virtually what the prototype, **??**, did, though on a very basic level. It defined a structure for an entry point of data, and an exit point in a scalable and versatile manner. There is no reason to doubt Reactive Blocks and OSGi on the point of data gathering, and they in fact also support control functionality, the next step after data gathering in ITS, with the shared and open data communications and modular structure. Making control modules based on inputs is really the same as making data gathering output modules with the exception of the way they handle the events. This is where the component system with small modules start to shine, since it is not likely that all ITS-stations will be in the same physical environment, or have the same input and output devices, customizable stations is a lot better than hoping to make a system that covers all fronts. Customization with Reactive Blocks and OSGi is as straight

forward as connecting to extra devices, but in stead of physically connecting wires you just install the needed bundles on the station.

7.1.2 Robustness

Making robust and safe applications is one of the strongest sides of Reactive Blocks. Because of its excellent verification tools, see section **??**, applications made in Reactive Blocks should in most cases be safer than those built with traditional tools. There is a much smaller chance of hitting deadlocks and experiencing race conditions, making the applications more robust. But though Reactive Blocks may be the main contributor to increased robustness, OSGi is not without merits either. When measuring experienced robustness, a requirement is often up-time. With OSGi's out of the box ability of updating components without experiencing any downtime whatsoever, even a dynamic component system where new functionality is added, and updated regularly can still support very good up-time. This is important when the systems require real-time data, and in ITS, examples of these are numerous, and include traffic lights, toll stations, etc.

7.1.3 Security

OSGi provides a security model that is based on the one in Java, that is strong. Security can really be implemented in any language, but it is rarely a good idea to do it yourself, since one small error can render the security useless and insecure. It is therefore good to be able to use tried and tested security implementations that are available in platforms with large communities. There would be few issues with making use of reusable 3rd party security implementations in OSGi. And in the unlikely event that there would not be OSGi bundles available, Java libraries are still usable in OSGi and Reactive Blocks, and should be more than capable of providing the security needed.

7.1.4 Extensibility

Component systems are made for extensibility. The use of a modular architecture is a way to incorporate extensibility into the development process from the start. Every module is an extension to a system, and following the principles of OSGi's no sharing when it is not absolutely necessary, extra modules add as little complexity to a system as possible, giving as much room for extensions as is viable. As the prototype, section **??**, explored in architectural choices, extensibility can be included from the first small modules.

7.2 Comparisons with rivalling technology choices

In this section, Reactive Blocks and OSGi is measured against the technologies explored in chapter **??**. Though finding out whether or not Reactive Blocks and OSGi are fit to solve a problem is important to whether or not they can be used, the question of them being a good choice can not be answered without measuring them up to their competitor. The goal of this section is to see Reactive Blocks and OSGi holds up compared to other technologies.

7.2.1 Java

Java is a great language, that is used in projects all over the world all the time. It has stood the test of time, and shows great merits. This is probably the main reason for Reactive Blocks and OSGi to be built on it. OSGi and Reactive Blocks are in essence a different way to write Java, as they both boil down to Java code, and bundles are .jar files that are run on the JVM. Most of the benefits of Java are also present in Reactive Blocks and OSGi, they benefit from the same performance, and the same platform independence. There are downsides to choosing Reactive Blocks and OSGi instead of vanilla Java however, there is not a lot of different tooling, and there is not yet a large developer community. When making Reactive Blocks modules it is a requirement that it is done in Eclipse, which is not a light-weight tool. Though in all fairness, there are no lightweight graphic model based development tools. More importantly than the tooling is the problem of a small community. It is not likely that there will be a lot of developers that are already very familiar with Reactive Blocks, and there will therefore have to be more time and resources allocated to familiarizing the developers with the platform than there would have been with Java, a higher initial cost.

7.2.2 Scala

Scala is a good scalable, and functional language, that incorporates the benefits of component systems. It is however, arguably, done even better by Reactive Blocks and OSGi, without the performance impairments from interpreting Scala for the JVM [**?**] [**?**]. Scala has some of the benefits of Java, and were the choice between the two, it would probably be very hard to say which was the best choice. Though since the possible technology of choice is between Scala and Reactive Blocks and OSGi the matter is different. Like with Java, Scala has the benefit of more diverse tooling (to a lesser extent than Java, but there is still a significant difference) and developers that don't need extra training to start developing. It does however, like Java, lack in the points of verification and support for building concurrent systems. Though Scala is built to be scalable, so is Reactive Blocks and OSGi, and it is hard to see that Scala does scalability any better.

7.3 Akka

Akka is probably the technology that is most similar to Reactive Blocks and OSGi that has been considered in this thesis. It is a less complex way to gain some of the benefits of OSGi, meaning less dependency issues, and the possibility of using more lightweight tooling. But it comes at a cost. The benefits of Reactive Blocks and OSGi, such as the improved verification, the visualization and the no down-time updating are all missing from Akka. In return, it supports both Java and Scala, and it can be used to make OSGi bundles[**?**]. This means that using Reactive Blocks and OSGi does not restrict future projects from choosing Akka, if Akka is not OSGi enabled, switching to OSGi is not as trivial.

7.4 Chapter Summary

In this chapter, Reactive Blocks and OSGi were measured against the challenges of ITS as well as against competing technologies. Both pros and cons have been discussed, with the key question of the thesis in mind, whether or not Reactive Blocks with OSGi is the right technology for developing an ITS. The points made in the discussion are used in chapter **??**.

Chapter 8
Conclusion and Further Work

In this chapter, an attempt to answer the main research question of the thesis is made, based on the findings of the report. Is Reactive Blocks with OSGi a good choice for developing ITS? What more can be done in the field is also elaborated upon in section **??**

8.1 Conclusion

In this section, an attempt to answer the research question of the thesis has been made. The question of whether Reactive Blocks with OSGi is a good choice of technology to build an ITS with will be answered by whether or not it is able to meet the challenges of ITS, see section **??**, and how it measures up to the rivalling technologies that are used widely in the field, see section **??**.

8.1.1 Meeting the Challenges

Reactive Blocks and OSGi are up to the task of meeting the challenges of ITS. Through the results from the prototype, and the documentation of Reactive Blocks and OSGi the capabilities of the technology has been evaluated. The prototype showed how one implementation architecture that fits the OSGi structure could host both data gathering applications at first, and traffic control applications as well once the time for those comes, in the roadside ITS sub-system. It does not come as a surprise that Reactive Blocks with OSGi is capable of meeting the challenges, but in addition to meeting the challenges they also show promise of meeting them well. There are in all likelihood better architectures than the one utilized in the prototype, section **??**, but even that one shows a scalable and simplistic system design that could host a plethora of functionalities both complex and simple in the same system without an exponential increase in complexity. There is great potential both for extensibility and robustness in the Reactive Blocks and OSGi platform, and they are more than capable of meeting the challenges of ITS.

8.1.2 Compared to the State of the Art

Reactive Blocks and OSGi are arguably a better choice of technology for ITS than the competition. In chapter **??** a few alternative technologies fit for building ITS with were explored, and in chapter **??** they were compared to Reactive Blocks and OSGi. The defining features of Reactive Blocks and OSGi that were not matched by any of the competitors were their visualization, verification and ability to update with no down-time. These are huge benefits when making systems that should control the potentially lethal traffic system where errors and down time in the worst case could reduce traffic safety of real living people. The package does not, however, come without its drawbacks. Reactive Blocks is a new technology that is fairly different from the usual tools of software building. It does not have a large knowledgeable following. It is therefore not likely that there will be as much possible reuse of 3rd party code as there would be using any of the other technologies explored in **??**. It also means that it will probably be hard to find experienced Reactive Blocks developers for the projects, so the ones to work on the project will probably need more training than with Java or Akka. This means that building an ITS in Reactive Blocks and OSGi will have a higher initial cost than if an ITS were to be built on any of the other mentioned technologies. Even with this downside, the benefits seem to outweigh the negatives. The cost of starting up is high, but it pays off by the decreased cost of fixing concurrency problems, and the low down-time that would be a likely result of using Reactive Blocks and OSGi.

8.1.3 Conclusion summarized

Reactive Blocks with OSGi is a very good technology for building an ITS. It is more than up to the task of meeting the challenges that must be met to make the software of a functioning ITS, and it rivals the competition in beneficial features. From the results of the research in the thesis, there is reason to think that Reactive Blocks with OSGi is a good, if not the best, choice for any ITS development project.

8.2 Further Work

In this thesis the platform of Reactive Blocks with OSGi has been explored and analysed with respect to ITS. This is only an initial step in the field of ITS. In this section, some further steps not yet taken are suggested.

8.2.1 Further Proof of Concept Research

The prototype described in the thesis shows a lot of promise, but it is not a complete field test for the Reactive Blocks and OSGi platform. Though it may be used as an indicator of how well Reactive Blocks and OSGi works in ITS, it can only stretch as far as the prototype environment. A larger scale application working in a

real-world environment, building upon the prototype of the thesis, would be a better approximation to a working ITS. This could highlight any problems not found during the work presented here.

8.2.2 Applying the Knowledge to a Working ITS

Another next step to this analysis is to put the results straight into practice. Making an ITS that at first supports data gathering, and makes the data available for research would be a natural next step to the work done in this project. Though some of the technology on the roads are not there at this point, there is still room for making the basics work, and updating with new and improved bundles as they come along. The first cars that support CAMs are for example not on the roads at the time of writing, but a module for CAM communication can be added to a roadside ITS-station at any time if it runs an OSGi environment.

8.2.3 How to use Traffic Data

There is still a lot of progress to be made in the field of traffic research. Gathering data in a traffic system is not of much value if the data is not used for anything. There must therefore be made an effort to find the best possible ways to increase traffic flow, safety, energy efficiency and the like using the data. This is what some of the extra functionality of the ITS will consist of, and where the real power of an ITS will lie.

References

[1] V. Cózar, J. Poncela, M. Aguilera, M. Aamir, and B. Chowdhry, "Cooperative vehicle-to-vehicle awareness messages implementation," in *Wireless Sensor Networks for Developing Countries*, ser. Communications in Computer and Information Science, F. Shaikh, B. Chowdhry, H. Ammari, M. Uqaili, and A. Shah, Eds. Springer Berlin Heidelberg, 2013, vol. 366, pp. 26–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41054-3_3

[2] "Osgi alliance website." [Online]. Available: http://www.osgi.org/Technology

[3] R. S. Hall, S. McCulloch, K. Pauls, and D. Savage, *OSGi in Action.* Manning Publications Co.

[4] S. Vegvesen, "Roadside its station specification," 2014.

[5] "Ecomove project." [Online]. Available: http://www.ecomove-project.eu/about-ecomove

[6] "Cooperative its messages for green mobility: An overview from the ecomove project." [Online]. Available: http://www.ecomove-project.eu/assets/Documents/Presentations/ITSVienna/EU-00628AlesianiLykkjaFestagBaldessari-v0.2.2.pdf

[7] "ecomove core technology integration." [Online]. Available: http://www.ecomove-project.eu/about-ecomove/subprojects/sp2/

[8] "Intelligent transport systems." [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/intelligent-transport

[9] W. H. Organization, "The top 10 causes of death," 2014. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs310/en/

[10] Ecobridge, "Causes of global warming," 2001. [Online]. Available: http://www.ecobridge.org/causes_of_global_warming.html

[11] "Osgi benefits." [Online]. Available: http://www.osgi.org/Technology/WhyOSGi

[12] "Keep it simple stupid." [Online]. Available: http://people.apache.org/~fhanik/kiss.html

[13] bitreactive, "Reactive blocks documentation: Essentials." [Online]. Available: http://reference.bitreactive.com/reference/essentials.html

[14] Bitreactive, "The secret twists to efficiently develop reactive systems," 2012. [Online]. Available: http://reference.bitreactive.com/papers/secret-twists.html

[15] "Etsi ts 102 637-2." [Online]. Available: http://www.etsi.org/deliver/etsi_ts/102600_102699/10263702/01.01.01_60/ts_10263702v010101p.pdf

[16] S. Bernstein and A. Blankstein. (2007) Key signals targeted, officials say. [Online]. Available: http://articles.latimes.com/2007/jan/09/local/me-trafficlights9

[17] M. E. Whitman and H. J. Mattord, *Principles of Information Security.* CENGAGE Learning, 2012.

[18] Raspberry pi. [Online]. Available: http://en.wikipedia.org/wiki/Raspberry_Pi

[19] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship.* Pearson Education Inc, 2009.

[20] Programming language popularity. [Online]. Available: http://www.langpop.com/

[21] Java gets four times faster with new symantec just-in-time compiler. [Online]. Available: http://grnlight.net/index.php/programming-articles/116-java-gets-four-times-faster-with-new-symantec-just-in-time-compiler

[22] The computer language benchmark game. [Online]. Available: http://benchmarksgame.alioth.debian.org/

[23] M. Odersky, P. Altherr, V. Cremet, D. I. Gilles Dubochet, B. Emir, S. McDirmid, S. Michelout, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger, "An overview of the scala programming language," 2006. [Online]. Available: http://www.scala-lang.org/docu/files/ScalaOverview.pdf

[24] What is akka? [Online]. Available: http://doc.akka.io/docs/akka/2.2-M2/intro/what-is-akka.html

[25] Typesafe case studies. [Online]. Available: http://www.typesafe.com/company/casestudies

[26] Why akka? [Online]. Available: http://doc.akka.io/docs/akka/2.2-M2/intro/why-akka.html

[27] (2012) Keeping borders safe with akka. [Online]. Available: http://downloads.typesafe.com/website/casestudies/Dutch-Border-Police-Case-Study-v1.3.pdf

[28] J. Bonér, "Introducing akka - simpler scalability, fault-tolerance, concurrency & remoting through actors," 2010. [Online]. Available: http://jonasboner.com/2010/01/04/introducing-akka/

[29] J. Faerman. (2012) Scala or java? exploring myths and facts. [Online]. Available: http://www.infoq.com/articles/scala-java-myths-facts

[30] (2012) Akka osgi support. [Online]. Available: http://doc.akka.io/docs/akka-modules/1.3.1/modules/osgi.html

Chapter A
Appendix I

A.1 The Prototype Modules

The prototype modules have been made available at github.com. They can be cloned from the github repository:

https://github.com/eivisand/reactive-ITS.git