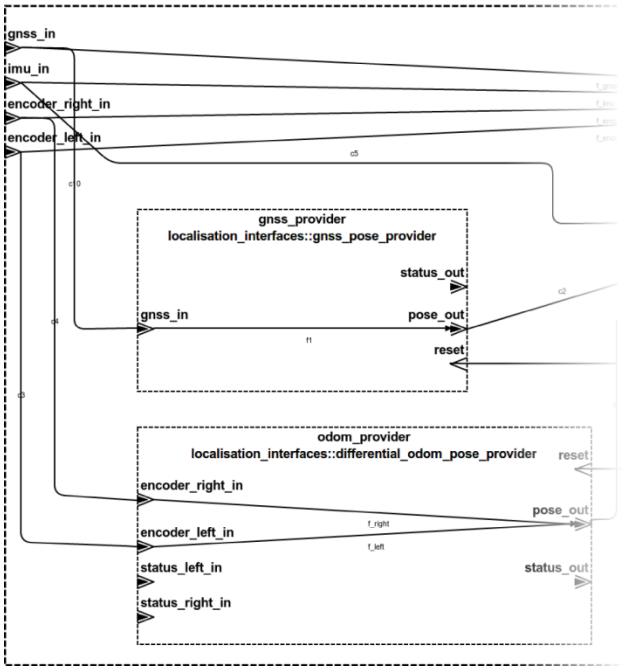
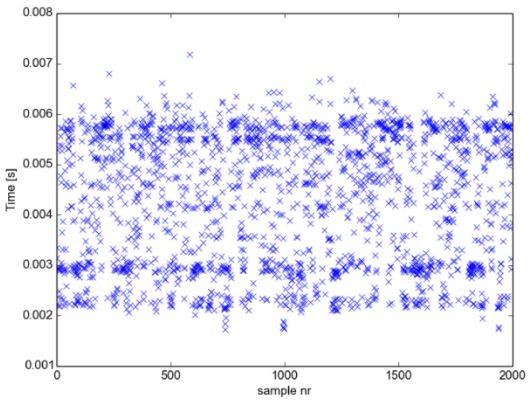
MODELLING FIELD ROBOT
SOFTWARE USING AADL

Electrical and Computer Engineering
Technical Report ECE-TR-25





system implementation localisation_subsystem.impl
subcomponents
utm_to_odom : **process** utm_to_odom_node.impl;
kalman_filter_node : **process** odom_imu_utm_ekf_node.impl;
differential_odometry : **process** differential_odometry_node.impl;
flows
f_gnss : flow path fmInformation_gpgga_tranmerc -> nc4 -> utm_to_odom.f_gnss -> nc6 -> kalman_filter_node.f_gnss -> nc9 -> fmKnowledge_pose;

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Modelling field robot software using AADL
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-25

Author: Morten Larsen
Department of Engineering – Electrical and Computer Engineering, Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2016 Pages: 41
Editing completed: April 2016

Abstract: This report contains a technical description and example on how robotic systems based on a distributed communication middleware can be modelled in AADL, incorporating hardware aspects. Furthermore analyses on the extra-functional properties such as bus-bandwidth and end-to-end latency are performed.

Keywords: robotics, modelling, Model-driven engineering, extra-functional properties, software engineering and systems

Supervisor: Rasmus Nyholm Jørgensen, Peter Gorm Larsen
Financial support: Conpleks Innovation ApS

Please cite as: Morten Larsen, Modelling field robot software using AADL, 2016 Department of Engineering, Aarhus University. Denmark. 41 pp. - Technical report ECE-TR-25

Cover image: Morten Larsen

ISSN: 2245-2087

MODELLING FIELD ROBOT SOFTWARE USING AADL

Morten Larsen
Aarhus University, Department of Engineering

Abstract

This report contains a technical description and example on how robotic systems based on a distributed communication middleware can be modelled in AADL, incorporating hardware aspects. Furthermore analyses on the extra-functional properties such as bus-bandwidth and end-to-end latency are performed.

Table of Contents

1

Introduction

The robotics domain is characterised by being a multi-disciplinary field. A robot can be decomposed into components, which each perform a function. These components may be both mechanical components, electrical components and software components. All these components and systems must be designed and implemented and work in cooperation in order for the robot to work. The components may be existing components which are bought and interfaced / integrated into a larger design, or they may be developed for the specific function required. The components of a robotic system fulfill a function, the complexity of the function may vary, and the requirements for the function may be specified in various degrees depending on completeness of the overall design. The main focus of this report is on the electrical components and the software components.

The requirements for these components may both be either functional and non-functional. The non-functional requirements is the main interest in this report. Timing is often a non-functional requirement for a robot. The timing requirements may be abstract, such as the time it takes from the user to press start, to the robot is executing the corresponding task, or a more concrete requirement such as the time it takes for a sensor-reading to reach a robot controller.

The design of the system and the decomposition of the system into subsystems and components may adversely affect the timing properties of the system. If for example a distributed hardware computing setup is designed, the bus has to be fast enough to support the communication between the components. From a software oriented view, the scheduling of the individual threads implementing the components, or the message passing overhead between components, may also affect the timing properties of the system.

1.1 Middlewares for robotic software components

The complexity of the software for robotics has resulted in multiple middlewares/frameworks which aims at reducing this complexity by providing abstractions such as a publish/subscribe message passing, component organisation/composistion features and other services. The Robot Operating System (ROS) [13] is a popular publish/subscribe based middleware for developing robotic systems and has been used in numerous robots [7, 9].

In ROS the component specification of inputs, outputs (subscribers, publishers) are embedded in the source language of the component which may be C++, python or other languages. The component specification can either be read by inspecting the source code or documentation manually, or starting the component and inspect the ROS graph, in order to view advertised publishers/subscribers and their type. The middleware does not typically provide tools that supports the design

of robotics systems, but are more focused on the implementation of these designs. Therefore there is a need for tools that supports the design of robotic systems, making architecture analysis, timing/performance analysis, experimentation with architecture alternatives, without having to resort to implementation languages. Such analyses can be done using general purpose tools such as spreadsheets and documents.

1.2 Modelling robotics systems

Model Driven Engineering (MDE) is becoming an established methodology for developing complex systems. MDE focuses on modelling the systems and then optionally generate large parts of the implementation by code-generation. The use of dedicated modelling languages allows the designer to incrementally detail the design with information and depending on the tools used, also validate the design against the requirements incrementally.

A model based approach to designing robotic systems can enable more analyses of the design before any commitment to a specific architecture is made, and may also increase the potential for reuse of software in different applications. A model based approach will allow the designer to select a desired level of abstraction depending on the context. For example in the early phase of design the overall architecture has to be established but the concrete detailed interfaces may not be specified yet, the modelling tools should support both abstract modelling and detailed modelling e.g data type assignment, unit specification, target system modelling and execution times.

1.3 Architecture Analysis and Design Language

Architecture Analysis and Design Language (AADL) is a modelling language specified by the Society of Automotive Engineers (SAE International) [14] for modelling architectures and performing detailed analyses of the architecture such as timing flows, scheduability, etc. AADL has a textual and a graphical notation, which allows the designer to select between textual models and graphical models depending on what is to be communicated. AADL supports both abstract and detailed modelling.

AADL has been used as base language for modelling or analysis activities within the aerospace domain [5, 12], automotive domain [17], robotics domain [1] and other Cyber-Physical Systems (CPSs) [10, 3]. This report uses the open source tool Open Source AADL Tool Environment (OSATE)[1] for developing and analysing the models of the Intelligent Marking robot.

1.4 Line Marking robot

The robot case used as basis for the modelling work done in this report is a Line Marking robot targeted outdoor grass based sport fields. The task is to paint the playing field with lines and arcs according to predefined dimensions. The robot navigates primarily using a Global Navigation Satellite System (GNSS) with Real Time Kinematics (RTK) corrections from a national grid. A 3D rendered model of the Intelligent Marking robot can be seen in Figure 1.1.

The performance of the Line Marking robot depends on the accuracy of the localisation system, navigation system, and lastly the variation of the delay from a position measurement is performed until the sprayer is activated. A large variation in the delay will result in inaccurate spray

[1] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2

Figure 1.1: Line Marking robot 3D render ©Intelligent Marking

activation and hence the lines may not be connected or may overlap, resulting in a poor quality of the painted field. For example a 20ms variation in the delay at a driving speed of 0.6m/s the start of the line may vary by 1.2cm, at 200ms the variation becomes 12cm, which is not acceptable.

Figure 1.2: Example of a sprayer delay which is not accounted for in the software, which in turn results in the overlap in the lower left corner of the figure.

1.5 Report overview

First, chapter 2 introduces the AADL modelling language, then the system is modelled using the abstract modelling features of AADL, finally the performance of the architecture with respect to sprayer activation latency is analysed. Then, chapter 3 present AADL models of the hardware platform and analyses the bus capacity as well as the hardware latency of the devices involved with the sprayer activation. Afterwards, chapter 4 present methods and examples of applying AADL modelling capabilities in order to model ROS based software components and infrastructure. Then chapter 5 validates some of the AADL models developed in the previous chapters through experiments, and present methods for obtaining the model parameters essential for performing the system latency analyses. Finally, chapter 6 summarises the findings and experiments performed in this report and provides concluding remarks.

2

Modelling robot software using AADL

In this chapter a common reference architecture is established based on the robot application introduced in chapter 1 using AADL. This section aims to demonstrate the high level modelling of an robotic application using AADL.

2.1 Introduction to AADL

The core elements of the Architecture Analysis and Design Language are components and property sets. A component is divided into two parts, a component type and one or more component implementations. A property set specifies properties which can be associated with a component classifier.

Component types represents the interface of a component furthermore classifies the component. AADL component classifiers falls into three categories, abstract, software and hardware. The `system` classifier can be used to group other components into a hierarchy, and the `abstract` classifier can be used in the early stage of a design to represent components whose classifier is yet to be determined. The interface of a component is described using features. A Feature can be a `port`, a `bus access` or others. Furthermore the component type has a compartment for specifying `properties`, `flows`, `prototypes` and others. The `flows` are specifications on which input features flows to which output features. The `prototypes` can be used to template a component type with other component types.

The component implementations, are concrete implementation of a component type. The implementation has a compartment for `subcomponents` which are used to further detail what components are actually used to implement a component type. The `subcomponents` can be empty in the case where there are no further elaboration to be done. The `flows` compartment in the component implementation, are used to elaborate the flows defined in the component type. The component implementation provides concrete paths for a flow through its subcomponents.

2.1.1 Abstract modelling

When modelling systems at an early phase, the main component classifiers used are `system` and `abstract`. A `system` represent a logical grouping of components or can represent black boxes. The `abstract` classifier is used for defining components which does not yet have a

more concrete classifier. The `abstract` classifier can also be used for abstract interfaces which requires further elaboration through subclassing.

The features of a component can be specified using the keyword `feature` which is again an abstract entity, the feature can later be refined to a concrete feature such as a `port`. The abstract feature can have a type as well as a direction. The subclassing component types can refine the feature using `refined to` keyword.

2.1.2 Software modelling

For modelling software aspects of a system, AADL provides `data`, `process`, `thread` and `subprogram` classifiers which are used to model software components in more detail. The `data` classifier can represent data types of varying complexity (integers, records and array). The `process` classifier represents a protected address space where the data of one or more threads can reside. The `thread` classifier represents a scheduable set of instructions. The thread modelling in AADL is primarily through `property sets`. The main properties for the `thread` classifier is found in `Thread_Properties` property set.

A `port` feature in AADL models exchange of data. The `port` feature has a direction of either `in`,`out` or `inout`. Furthermore a port can be of either `event port`, `data port` or `event data port`. An `event port` in AADL models an event which can be sent or received to/from the component, the event may carry data. The `data port` models exchange of data, either via a shared variable or by message passing. The data is not queued, meaning the current value is always read. The `event data port` represents message passing with queuing of messages.

2.1.3 Hardware modelling

AADL has hardware related component classifiers such as `bus`, `memory`, `processor` and `device`. The `processor` represents the hardware which can execute instructions from threads and provide the interface to the memory and busses. Processes and threads are assigned to a specific processor by using the `Actual_Processor_Binding` property. A `device` classifier represents a device where the logical data exchange is done via the features, but the actual exchange of data between devices is occurring via a `bus`. A `bus` represents a communication channel where data can be exchanged. Devices and processors communicates with each other using busses. Feature connections between a device and a processor can be bound to a specific bus by using the `Actual_Connection_Binding` property.

2.2 Abstract Modelling of the Intelligent Marking robot

Depending on the type of project, it may be beneficial to start modelling the overall system architecture and its components in terms of subsystem black-boxes which are to be refined. If the project depends on a specific critical feature, it may be beneficial to start with a bottom up approach for modelling the system, by starting with exploring the design of the critical feature, and then later model the system as a whole. For the Intelligent Marking robot, we start with modelling the system and its subsystems. The robot platform is first modelled as a generic robot platform with the sprayer seen as an application specific part. Reuse of components between robot platforms is common and is also a goal. The generic architecture (reference architecture) is first established based on previous projects and experience. Then the component library is developed by modelling the interface of the already developed components in a generic way.

2.2.1 Establishing a reference architecture

The definition of a reference architecture can be done in several ways, one can model the concrete applications first and then extract commonalities between the applications into a reference architecture, or the reference architecture can be established and refined as the individual applications are being modelled, as also suggested in [4]. Several field robot software architectures such as Agriture [11], Hortibot/AgroBot [8], and FroboMind [7] exists and have been demonstrated on various platforms. In this work we are inspired by the overall architecture in Frobomind, but do not fully adhere to it.

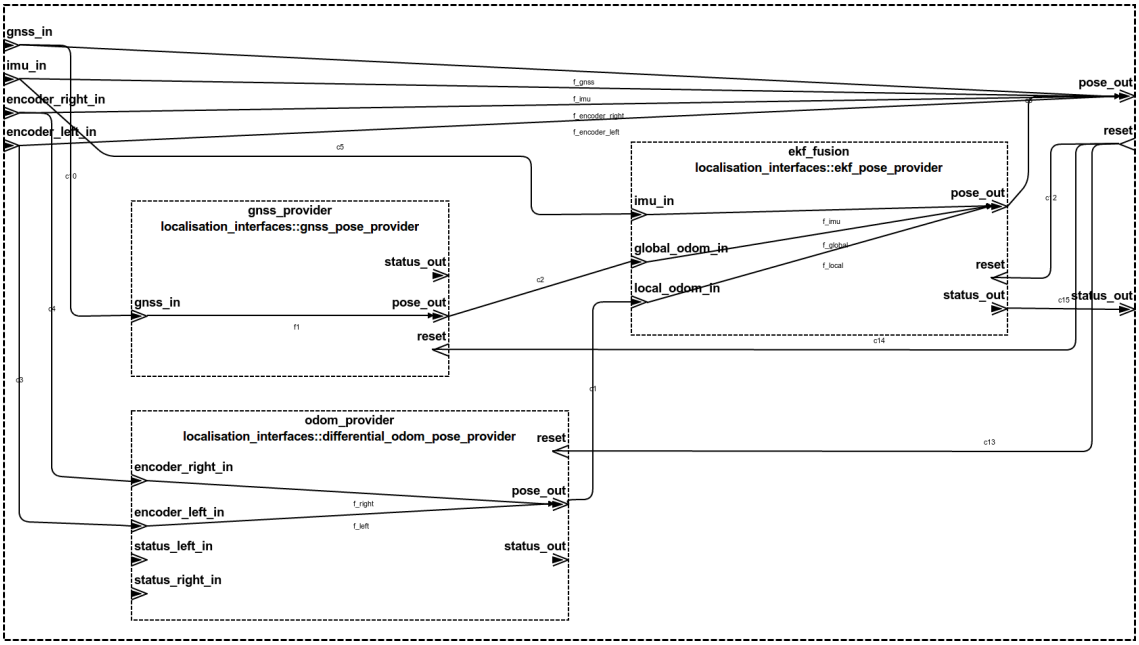


Figure 2.1: AADL Graphical model of a design pattern for localisation using RTK and local odometry, shown with AADL flow specifications.

In AADL the reference architecture can be modelled as abstract components having predefined ports with the type of the ports being unspecified, or alternatively typed by a prototype specification. The aim of the reference architecture is to establish the overall hierarchy of components and their type, as well as define common patterns for composition of these component types. This may be a common pattern for a differential drive robot where each motor must monitor the other motor for errors and act accordingly. In Figure 2.1, the reference design for a localisation system using RTK GNSS and local odometry is presented. This reference design is used when designing the concrete localisation system for the Line Marking robot.

2.2.2 Component library

This section describes the modelling of a component library in AADL which will contain the general components used across multiple applications, the separation of generic components into a library facilitates reuse. Listing 2.1 shows a partial view of the interfaces defined in the `interfaces::localisation` package. A graphical view of the package can also be seen in Figure 2.2.

```
package interfaces::localisation
public
abstract pose_provider
  features
```

```
    pose_out : out event data port;
    reset : in event data port;
    status_out : out event data port;
end pose_provider;

abstract gnss_pose_provider extends pose_provider
  features
    gnss_in : in event data port;
  flows
    f1 : flow path gnss_in -> pose_out;
end gnss_pose_provider;

abstract differential_odom_pose_provider extends pose_provider
  features
    encoder_left_in : in event data port;
    encoder_right_in : in event data port;
    status_left_in : in event data port;
    status_right_in : in event data port;
  flows
    f_left : flow path  encoder_left_in ->pose_out;
    f_right : flow path  encoder_right_in ->pose_out;
end differential_odom_pose_provider;

end interfaces::localisation;
```

Listing 2.1: Abstract interface definitions of the localisation components

The modelling mechanisms provided by AADL enables us the create interfaces which other components can inherit, thereby allowing them to be used as plug-in replacements when detailing the reference architecture.

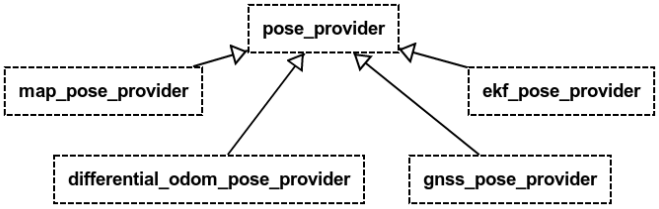


Figure 2.2: Graphical overview of the `interfaces::localisation` package

In the high level modelling of the architecture and components, we use `abstract` component types, such that the decision of a component being a `thread` or `process`, or something else is deferred until the concrete platform is determined.

2.2.3 System model of the Intelligent Marking robot

With the reference architecture and the component library in place, the application specific instance of the generic architecture and the components can be developed. In the process of refining the reference architecture, components which are missing can be identified and a specification derived from the AADL model as input for the component developer.

2.3 Analysing the system models

The flow latency from a sensor reading is acquired from the GNSS, until the sprayer is activated, is a key performance parameter for the line marking robot. The sprayer module needs to know this latency in order to mark the lines precisely, furthermore we need to determine the variation

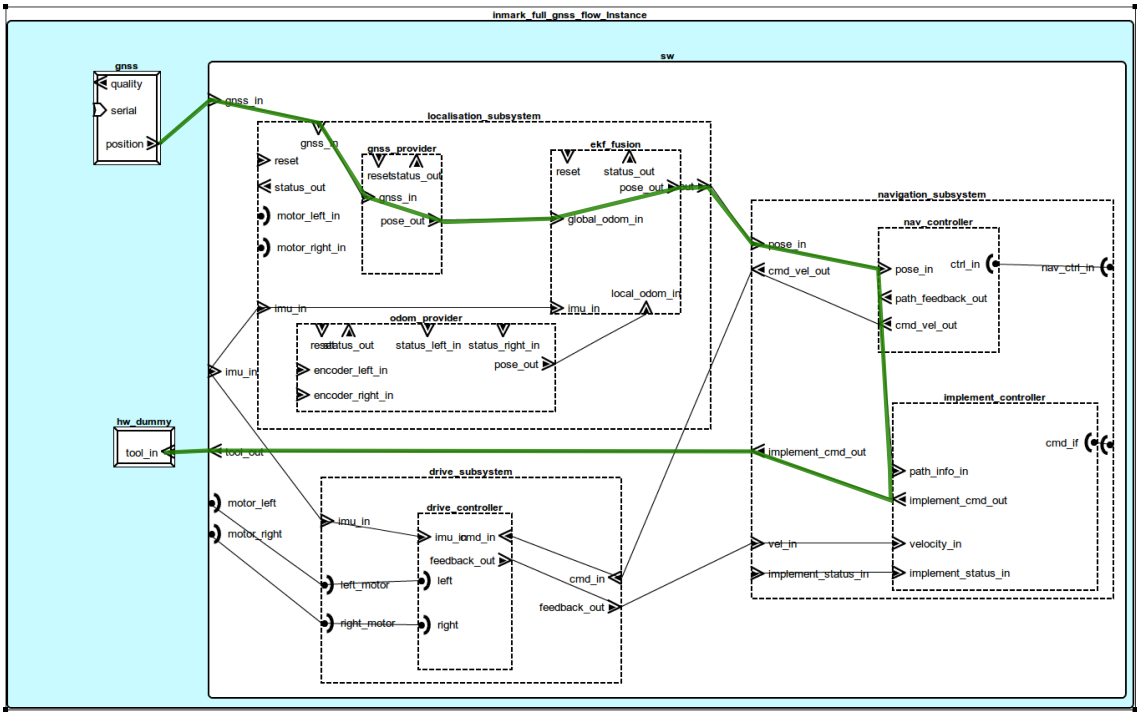


Figure 2.3: AADL system implementation instance using the reference architecture and interfaces with the flow path from the GNSS to the sprayer highlighted in green.

of this delay. An initial estimation of the flow latency is done using the Flow Latency Plugin in OSATE and the flow constructs of the AADL language. The flow latency analysis can be useful in establishing constraints on the design of the system, and as the implementation of the individual components takes place, the expected flow latency will be compared with the actual, and warnings/errors will be reported.

```
system implementation inmark_full.gnss_flow
  subcomponents
    sw : system robots::ci_base_robot::generic_rtk_gnss_robot_diff_drive.std;
    gnss : device devices::gnss::gnss_leica_mojo3_v2;
    hw_dummy : device dummy;
  connections
    c1: port gnss.position -> sw.gnss_in;
    c2 : port sw.tool_out -> hw_dummy.tool_in;
  flows
    f_gnss_lat :   end to end flow gnss.output -> c1 ->
      sw.f_gnss_tc -> c2 -> hw_dummy.input
      {Latency => 1 ms .. 12 ms;};
  properties
    Compute_Execution_Time => 1100 us .. 1800 us applies to
      sw.localisation_subsystem.gnss_provider;
    Compute_Execution_Time => 2 ms .. 3 ms applies to
      sw.localisation_subsystem.ekf_fusion;
    Compute_Execution_Time => 500 us .. 800 us applies to
      sw.navigation_subsystem.implement_controller;
    Period => 50 ms applies to
      sw.localisation_subsystem.ekf_fusion;
    Period => 100 ms applies to
      sw.localisation_subsystem.gnss_provider;
    Period => 20 ms applies to
      sw.navigation_subsystem.nav_controller,
      sw.navigation_subsystem.implement_controller,
      hw_dummy;
end inmark_full.gnss_flow;
```

Listing 2.2: Initial flow specification and allocation of periods and computation time for each component

The numbers used in Listing 2.2 are based on "best guesses", and are merely used for demonstration purposes. These numbers may eventually be based on datasheet information, or information from past usage of the components or other methods presented in chapter 5. The key is to demonstrate that the rather abstract model presented in Listing 2.2 can provide the designer with insight into the performance of the chosen architecture. Furthermore it can be used to evaluate design alternatives.

Table 2.1 shows the resulting flow analysis report with some fields left out. The report details the individual latency contributions made by components and connections along the specified flow in Listing 2.2. The latency analysis report provides the total minimum actual latency and the total maximum actual latency, which is then compared with the specified one. Table 2.1 there are multiple entries per latency contributor. Depending on the properties of a component, there will be an entry for sampling latency, queuing latency and processing latency.

As seen from Table 2.1, the jitter is $220ms - 5ms = 215ms$ Which will result in a line start/end variation of almost $13cm$ at $0.6m/s$ driving speed which is above the tolerated deviation. The main contribution of the variation in the latency is the `gnss_provider`, which executes at $100ms$ period. The reason for this choice is that this is the frequency the GNSS device provides data at. Since the `gnss_provider` is periodic this will at worst case mean that it has just executed before the new sample arrived from the GNSS device. In order to reduce the variation in the design, we can increase the period of the `gnss_provider` or make it an Aperiodic thread which triggers when it receives a message from the GNSS device. We can explore this alternative in AADL quite easily by creating a new `system implementation` which extends the original instance shown in Listing 2.2. This extension is shown in Listing 2.3 with the results of the latency analysis shown in Table 2.2.

Contributor	Min Value	Min Method	Max Value	Max Method
device gnss	0.0ms	first sampling	0.0ms	first sampling
device gnss	1.0ms	specified	3.0ms	specified
Connection gnss.position	0.0ms	no latency	0.0ms	no latency
abstract sw.localisation_subsystem.gnss_provider	0.0ms	sampling	100.0ms	sampling
abstract sw.localisation_subsystem.gnss_provider	1.1ms	processing time	1.8ms	processing time
Connection gnss_provider.pose_out	0.0ms	no latency	0.0ms	no latency
abstract sw.localisation_subsystem.ekf_fusion	0.0ms	sampling	50.0ms	sampling
abstract sw.localisation_subsystem.ekf_fusion	2.0ms	processing time	3.0ms	processing time
Connection localisation_subsystem.ekf_fusion.pose_out	0.0ms	no latency	0.0ms	no latency
abstract sw.navigation_subsystem.nav_controller	0.0ms	sampling	20.0ms	sampling
abstract sw.navigation_subsystem.nav_controller	0.0ms	no latency	0.0ms	no latency
Connection nav_controller.path_feedback_out	0.0ms	no latency	0.0ms	no latency
abstract sw.navigation_subsystem.implement_controller	0.0ms	sampling	20.0ms	sampling
abstract sw.navigation_subsystem.implement_controller	0.5ms	processing time	0.8ms	processing time
Connection sw.navigation_subsystem.implement_controller.implement_cmd_out	0.0ms	no latency	0.0ms	no latency
device hw_dummy	0.0ms	sampling	20.0ms	sampling
device hw_dummy	0.0ms	queued	1.1ms	queued
device hw_dummy	0.5ms	specified	1.1ms	specified
Latency Total	5.1ms		220.8ms	

Table 2.1: Flow latency report for the model in Listing 2.2 using the OSATE Flow Analysis Plugin, shown as individual contributions each component and connection makes to the overall latency.

From the results presented in Table 2.2 it can be seen that the variation in the latency has been reduced to $120ms - 5ms = 115ms$ Which will result in a line start/end variation of almost $7cm$ at $0.6m/s$ driving speed which is closer to the accepted deviation. From here on the designer can continue to improve the latency by experimenting with periods of components or the dispatch protocol. When the abstract classifiers are replaced with either thread or process classifiers, the CPU load can be estimated using the OSATE plugin. Alternatively the budget for each component can be specified using the `SEI::MIPSBudget` property.

```
system implementation inmark_full.gnss_flow2 extends inmark_full.gnss_flow
  properties
    Compute_Execution_Time => 1100 us .. 1800 us applies to
      sw.localisation_subsystem.gnss_provider;
    Compute_Execution_Time => 2 ms .. 3 ms applies to
      sw.localisation_subsystem.ekf_fusion;
    Compute_Execution_Time => 500 us .. 800 us applies to
      sw.navigation_subsystem.implement_controller;
    Period => 50 ms applies to
      sw.localisation_subsystem.ekf_fusion;
    Period => 0 ms applies to
      sw.localisation_subsystem.gnss_provider;
    Dispatch_Protocol => Aperiodic applies to
      sw.localisation_subsystem.gnss_provider;
    Period => 20 ms applies to
      sw.navigation_subsystem.nav_controller,
      sw.navigation_subsystem.implement_controller,
      hw_dummy;
end inmark_full.gnss_flow2;
```

Listing 2.3: Listing of a system implementation in AADL with modified scheduling properties of the gnss_provider component compared to Listing 2.2

Contributor	Min Value	Min Method	Max Value	Max Method
device gnss	0.0ms	first sampling	0.0ms	first sampling
device gnss	1.0ms	specified	3.0ms	specified
Connection gnss.position	0.0ms	no latency	0.0ms	no latency
abstract sw.localisation_subsystem.gnss_provider	0.0ms	sampling	0.0ms	sampling
abstract sw.localisation_subsystem.gnss_provider	1.1ms	processing time	1.8ms	processing time
Connection gnss_provider.pose_out	0.0ms	no latency	0.0ms	no latency
abstract sw.localisation_subsystem.ekf_fusion	0.0ms	sampling	50.0ms	sampling
abstract sw.localisation_subsystem.ekf_fusion	2.0ms	processing time	3.0ms	processing time
Connection localisation_subsystem.ekf_fusion.pose_out	0.0ms	no latency	0.0ms	no latency
abstract sw.navigation_subsystem.nav_controller	0.0ms	sampling	20.0ms	sampling
abstract sw.navigation_subsystem.nav_controller	0.0ms	no latency	0.0ms	no latency
Connection nav_controller.path_feedback_out	0.0ms	no latency	0.0ms	no latency
abstract sw.navigation_subsystem.implement_controller	0.0ms	sampling	20.0ms	sampling
abstract sw.navigation_subsystem.implement_controller	0.5ms	processing time	0.8ms	processing time
Connection sw.navigation_subsystem.implement_controller.implement_cmd_out	0.0ms	no latency	0.0ms	no latency
device hw_dummy	0.0ms	sampling	20.0ms	sampling
device hw_dummy	0.0ms	queued	1.1ms	queued
device hw_dummy	0.5ms	specified	1.1ms	specified
Latency Total	5.1ms		120.8ms	

Table 2.2: Flow latency report for the model in Listing 2.3 using the OSATE Flow Analysis Plugin, shown as individual contributions each component and connection makes to the overall latency.

3

Modelling robot hardware

Modelling the hardware can be beneficial for more detailed analysis, as well as providing an overview of how the software interacts with the hardware. Figure 3.1 show the hardware subsystem of the Intelligent Marking robot. There are two primary analyses we wish to carry out, one is obtaining an estimate of the bus bandwidth used/required by the hardware devices, secondly we want to estimate the latency of the hardware subsystem.

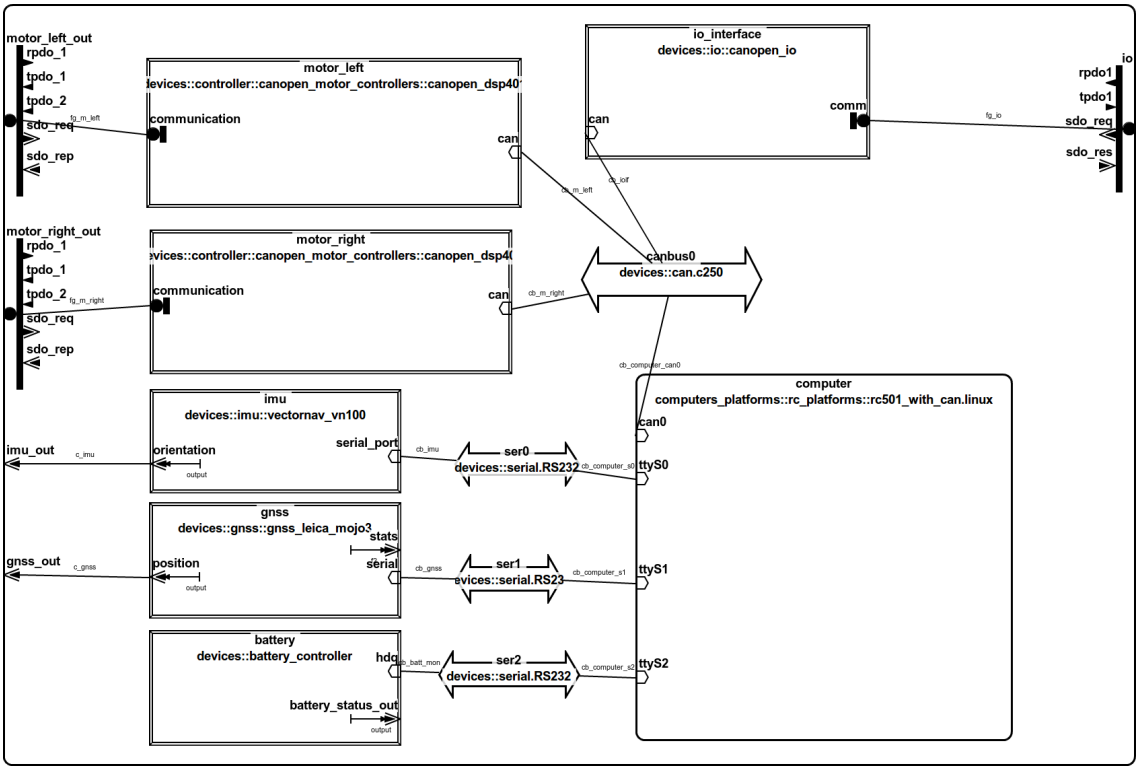


Figure 3.1: Graphical view of an AADL system implementation of the Intelligent Marking robot hardware platform

The hardware is based on a single computing platform with a CAN bus connection to the motors and a generic IO device for sprayer activation and auxiliary functions such as buttons and lights. The GNSS and the Inertial Measurement Unit (IMU) each communicates over their own

serial port. The hardware model shows both the physical busses present, but also shows the ports provided/required by the devices from a software point of view.

3.1 Modelling communication busses

The CAN bus is modelled as a `bus` in AADL. Listing 3.1 shows the AADL textual model.

```
bus can
end can;
bus implementation can.c125
  properties
    SEI::BandWidthCapacity => 125000.0 bitsps;
    Latency => 1 us .. 1084 us;
    Transmission_Time => [ Fixed => 256 us .. 256 us;
      PerByte => 64 us .. 88 us; ];
end can.c125;
```

Listing 3.1: AADL model of the CAN bus with a bitrate of 125000 bits/s

The AADL property set `Communication_Properties` enables the modeller to specify the transmission time of the bus in terms of a fixed value representing constant overhead, and a per byte transmission time. Calculating the transmission time for a CAN bus, can be quite complex due to the way frames are sent over the bus. The CAN bus hardware may add extra bits to the frame depending on the contents of the frame (bit stuffing, if more than 6 consecutive bits with value 1 is encountered). For obtaining a reasonable estimate we have used an online calculator[1] found at to provide values for the transmission time. The fixed transmission time in our model does not include the id part of the CAN frame, but instead it is assumed to be part of the data transmitted. This allows us to model both normal 11bit identifiers and 29bit identifiers using the same CAN bus model.

The motors and IO devices are using the CANopen protocol, which specifies both hardware properties (Connectors, Cabling) and protocols on top of the CAN bus. The CANopen protocol is modelled in detail in order to better estimate the bandwidth used by the hardware. Listing 3.2 shows how the CANopen protocol is modelled using feature groups to group the communication between a CANopen compliant motor and driver, and shows an example of the data being transmitted over the bus.

```
feature group canopen_dsp401_motor_plug
  features
    rpdo_1 : in event data port canopen_dsp401_pdo.r1;
    tpdo_1 : out event data port canopen_dsp401_pdo.t1;
    tpdo_2 : out event data port canopen_dsp401_pdo.t2;
    sdo_req : in event data port canopen_dsp401_sdo.i;
    sdo_rep : out event data port canopen_dsp401_sdo.i;
end canopen_dsp401_motor_plug;

data implementation canopen_dsp401_pdo.t1
  subcomponents
    cob : data Base_Types::Unsigned_32;
    status_word : data Base_Types::Unsigned_16;
  properties
    Data_Size => 6 Bytes;
end canopen_dsp401_pdo.t1;
```

Listing 3.2: Partial AADL model of the CANOpen protocol

[1] www.esacademy.com/en/library/calculators/can-best-and-worst-case-calculator.html

AADL allows us to model the semantic connection between a device and the software executing on a processor, and then later specify on which bus the communicated data (Connections between device ports and thread ports) flows using the `Actual_Connection_Binding` property. Other properties of the hardware which could be beneficial to include in the model could be the power requirements for the individual devices, in order to establish the total power usage requirements as done in [16].

3.2 Bus bandwidth analysis

In order to fully analyse the bus bandwidth utilised, we introduce a set of dummy drivers to the hardware model in Figure 3.1, these dummy drivers models the commands sent to the motors and the commands sent to the IO box. Figure 3.2 shows the modified model of the hardware as an instance view.

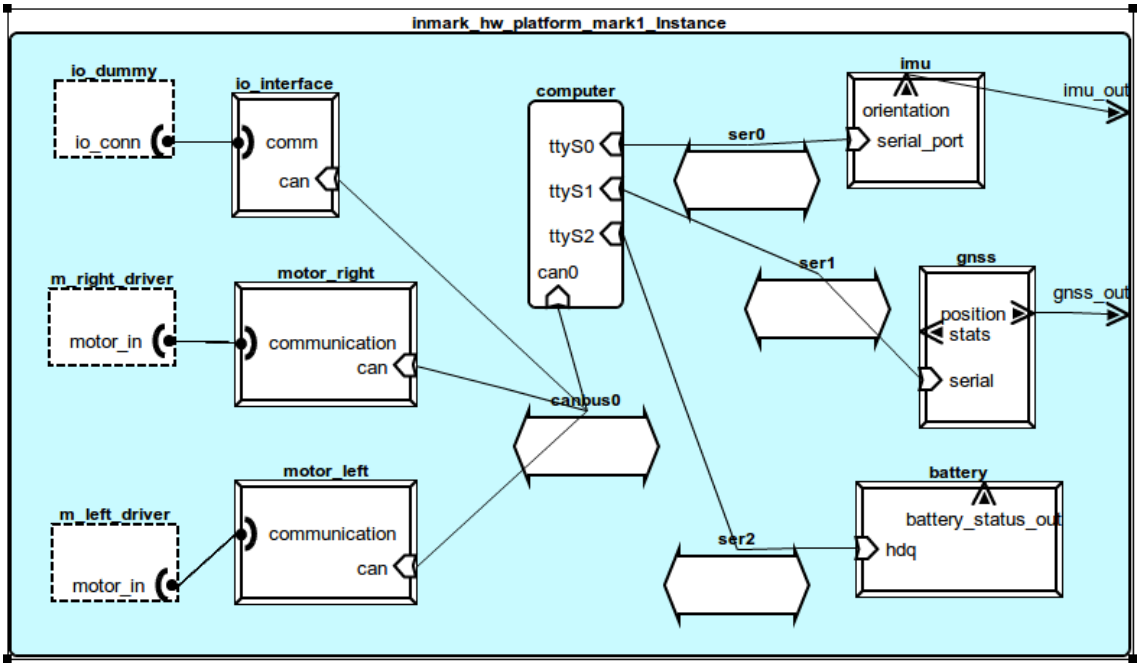


Figure 3.2: AADL system implementation of the Intelligent Marking robot hardware platform

The instance shown in Figure 3.2 is analysed using the OSATE Bus Bandwidth analysis tool, in order to estimate the bus utilisation. Table 3.1 shows the usage report for the CAN bus (`canbus0`), Table 3.2 and Table 3.3 shows the bandwidth usage for the `ser0` and `ser1` busses from Figure 3.2.

As seen from Table 3.1, the total bandwidth usage is 2.92 Kilo bytes per second, which is well below the maximum bandwidth of 15.625 KByteps available.

Similarly the usage of the `ser0` and `ser1` busses are well below the maximum of the serial RS232 bus bandwidth of 11.52 kilo bytes per second.

Source	Actual
m_left_driver.motor_in.rpdo_1	0.36 KBytesps
m_left_driver.motor_in.sdo_req	0.08 KBytesps
m_right_driver.motor_in.rpdo_1	0.36 KBytesps
m_right_driver.motor_in.sdo_req	0.08 KBytesps
io_dummy.io_conn.rpdo1	0.36 KBytesps
io_interface.comm.tpdo1	0.36 KBytesps
motor_left.communication.tpdo_1	0.1 KBytesps
motor_left.communication.tpdo_2	0.48 KBytesps
motor_left.communication.sdo_rep	0.08 KBytesps
motor_right.communication.tpdo_1	0.1 KBytesps
motor_right.communication.tpdo_2	0.48 KBytesps
motor_right.communication.sdo_rep	0.08 KBytesps
Total 2.920 KBytesps	Available 15.625 KBytesps

Table 3.1: OSATE Bus bandwidth report for the canbus0 where the IO box and the two motors communicate with the computer.

Source	Actual
imu.orientation	1.2 KBytesps
Total 1.200 KBytesps	Available 11.52 KBytesps

Table 3.2: Bus bandwidth report for the ser0 bus where the IMU communicates with the computer.

Source	Actual
gnss.position	0.6 KBytesps
Total 0.6 KBytesps	Available 11.52 KBytesps

Table 3.3: Bus bandwidth report for the ser1 bus where the GNSS communicates with the computer.

3.3 Hardware flow latency analysis

The hardware model from Figure 3.1 is extended with some flow specifications, which allows us to perform flow analysis. The hardware contribution to the variation in the latency of the sprayer module is investigated. In this section it is assumed that we do not have any knowledge of how the software is structured or performs, in order to show how AADL can be used in early design phases to estimate latencies for hardware architectures. Listing 3.3 shows the flow specifications for the hardware model from Figure 3.2. The properties listed in Listing 3.3 assigns guesstimates to the hardware devices internal operating latencies, these number may come from information from the manufacturer, or may have to be guessed. In this case we assume that the internal period of the IO interface and GNSS device is $1ms$. Furthermore we assign a computation time to the devices reflecting the fact that from the input is sampled in the device until a result produced time passes.

```
flows
  f_io_cmd : end to end flow io_dummy.cmd ->
    fg_io -> io_interface.cmd_in;

  f_gnss_lat : end to end flow gnss.output ->
    c_gnss -> gnss_dummy.input;
properties
  Period => 1 ms applies to io_interface;
  Period => 1 ms applies to gnss;
  Compute_Execution_Time => 200 us .. 300 us applies to io_interface;
  Compute_Execution_Time => 200 us .. 300 us applies to gnss;
```

Listing 3.3: Flow specification for the hardware flows involved in the overall GNSS to Sprayer flow

The transmission time properties assigned to the CAN bus is the same as shown in Listing 3.2 and the transmission time properties assigned to the serial bus connecting the GNSS with the computer is shown in Listing 3.4

```
bus implementation serial.RS232
  properties
    SEI::BandWidthCapacity => 11520.0 Bytesps;
    Latency => 10 us .. 10 us;
    Transmission_Time => [ Fixed => 1 us .. 1 us;
                           PerByte =>  87 us .. 87 us; ];
end serial.RS232;
```

Listing 3.4: AADL bus model of the serial RS232 connection between the GNSS device and the computer

The results of the latency flow analysis for the IO interface device is shown in Table 3.4. The variation of the delay is $1.4ms$ which is an insignificant contribution the sprayer delay.

Contributor	Min	Min Method	Max	Max Method
abstract io_dummy	0.0ms	first sampling	0.0ms	first sampling
abstract io_dummy	0.0ms	no latency	0.0ms	no latency
Connection	1.0ms	no latency	1.3ms	no latency
device io_interface	0.0ms	sampling	1.0ms	sampling
device io_interface	0.2ms	processing time	0.3ms	processing time
Latency Total	1.2ms		2.6ms	

Table 3.4: OSATE Flow Latency Analysis Plugin report for the IO interface device flow, from the device to the abstract driver.

The results of the latency flow analysis for the GNSS interface device is shown in Table 3.5 and again the variation in the latency is insignificant for the sprayer delay. However if a different bus was selected for interfacing the computer with the sprayer actuation hardware, the delays may have been significant.

Contributor	**Min**	**Min Method**	**Max**	**Max Method**
device gnss	0.0ms	first sampling	0.0ms	first sampling
device gnss	0.2ms	processing time	0.3ms	processing time
Connection	2.6ms	no latency	2.6ms	no latency
abstract gnss_dummy	0.0ms	sampling	0.0ms	sampling
abstract gnss_dummy	0.0ms	no latency	0.0ms	no latency
Latency Total	2.8ms		2.9ms	

Table 3.5: OSATE Flow Latency Analysis Plugin report for the GNSS device flow from the GNSS device to the abstract driver.

4

Modelling software components

In this section, we present a methodology for modelling robotic software components in AADL. ROS [13],Orocos [2],SmartMDSD [15] and OPRoS [6] are all based on a publish/subscribe system with varying complexity and feature sets. However the middlewares have three interaction patterns in common. The first is the basic publish/subscribe pattern. The second is the Remote Procedure Call (RPC) style interaction pattern. The third is a mechanism for declaring component parameters, which must be supplied (assigned) by the component user. We wish to be able to model these three styles of component interaction in AADL. First a set of common "patterns" found in ROS node implementation is presented and a corresponding AADL model is developed. Then each section elaborates on a specific set of constructs found in the ROS middleware and presents a corresponding AADL model.

4.1 Anatomy of a ROS based node

A typical pattern of an implementation of a ROS node component can be seen in Listing 4.1

```
// Read parameters from server
// setup outputs
// setup inputs
void differential_drive::configure()
{
  configureFromParameters();
  configurePublishers();
  configureSubscribers();
  nh.createTimer(ros::Duration(0.1),
    &differential_drive::onMainLoopTimer, this);
}
// Callback function for the CmdMsg topic
void differential_drive::onCmdMsg(
  const geometry_msgs::TwistStamped::ConstPtr& msg)
{
  // read relevant fields from message into internal data members
}

// Timer callback function called every period
void differential_drive::onMainLoopTimer(const ros::TimerEvent&)
{
  // read internal data members

  // update controllers / state

  // write outputs
```

```
}
```

Listing 4.1: Sample c++ pseudo code of a periodically triggered ROS node

The pattern shown in Listing 4.1 is a periodically triggered node, which samples its inputs (regardless of when they were actually received) and produces an output at each period. We refer to this pattern as a periodically sampled node. A second pattern is a asynchronous pattern, where the callback for the received messages on a topic directly triggers the algorithm and produces an output. This pattern is shown in Listing 4.2.

```
// Read parameters from server
// setup outputs
// setup inputs

void differential_drive::configure()
{
  configureFromParameters();
  configurePublishers();
  configureSubscribers();
}
// Callback function for the CmdMsg topic
void differential_drive::onCmdMsg(
  const geometry_msgs::TwistStamped::ConstPtr& msg)
{
  // read the message
  // run algorithm
  // write outputs
}
```

Listing 4.2: Sample c++ pseudo implementation asynchronously triggered ROS node

A hybrid between the two patterns presented so far (periodically sampled and the asynchronous) exists. A node may need to perform partial processing of the received data in the callback function. As an example consider a node which receives inputs from two encoders and produces an estimate of the odometry. In this case the encoders are publishing delta readings and therefore loss of messages results in inaccurate odometry calculations. The node has to synchronise the two topics by accumulating each received message until both callback has received data. Listing 4.3 shows an example of such a node. We refer to this kind of pattern as a hybrid.

```
// Callback function for the right encoder
void differential_odom::onEncRightMsg(const msgs::encoder::ConstPtr& msg)
{
  // accumulate ticks and signal to timer thread that we have data
}

// Callback function for the left encoder
void differential_odom::onEncLeftMsg(const msgs::encoder::ConstPtr& msg)
{
  // accumulate ticks and signal to timer thread that we have data
}

// Timer callback function called every period
void differential_odom::onMainLoopTimer(const ros::TimerEvent&)
{
  // check if left and right has produced data
  // calculate odometry if both has produced data
  // else wait another period
}
```

Listing 4.3: Sample c++ pseudo implementation of a hybrid ROS node

4.2 Modelling scheduling properties

The corresponding AADL model of the periodic and asynchronous ROS node are shown in Listing 4.4. The `Thread_Properties::Dispatch_Protocol` and `Timing_Properties::Period` properties are used to model the timer and the triggering of a callback when a message is received.

```
thread periodic_component
  features
    cmd_msg_in : in  event data port
      middleware::ros::geometry_msgs::TwistStamped;
  properties
    Dispatch_Protocol => Periodic;
    Period => 100 ms;
end periodic_component;

thread asynchronous_component
  features
    cmd_msg_in : in  event data port
      middleware::ros::geometry_msgs::TwistStamped;
  properties
    Dispatch_Protocol => Aperiodic;
    Dispatch_Trigger => (reference(cmd_msg_in));
end asynchronous_component;
```

Listing 4.4: AADL model of a periodic and asynchronous ROS node

In Figure 4.1 as graphical view of the AADL model of a hybrid ROS node is shown. The model features both normal port connections and data access connections which indicates that data is shared between the threads in the thread group. This detailed modelling can be useful if the component is to work both in non-preemptive and preemptive situations. The default for the ROS node is to work in a single threaded mode, where all events (Timer events, Message reception) share a single callback queue thread.

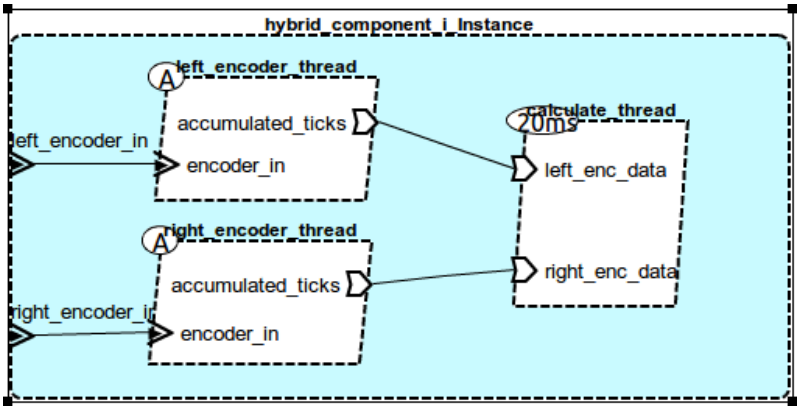


Figure 4.1: AADL instance of a thread group modelling the hybrid ros node from Listing 4.3

4.3 Communication interfaces

For the communication interface part, that is, the publish/subscribe pattern and the RPC pattern we can directly use the `port` feature. A publisher is modelled as a `out event data port` feature of an AADL component type. The `event data` specifies that the individual received messages are queued.

```
thread gnss_localisation extends interfaces::localisation::gnss_pose_provider
  features
    gnss_in : refined to  in event data port middleware::ros::msgs::gpgga_tranmerc;
    pose_out : refined to out event data port middleware::ros::nav_msgs::Odometry;
    reset : refined to in event data port middleware::ros::std_msgs::Bool;
    status_out :refined to out event data port middleware::ros::std_msgs::Bool;
end gnss_localisation;

thread implementation gnss_localisation.std
  subcomponents
    config : data gnss_localisation_config.impl;
  properties
    Dispatch_Protocol => Periodic;
 end gnss_localisation.std;
```

Listing 4.5: Sample of an ROS node modelled as a thread

The `thread implementation` in Listing 4.5 specifies that this thread is triggered periodically, which also implies that the inputs are sampled at each trigger. This model is a simplified view on the actual implementation the component. In ROS a number of threads are created for each node, these threads are used for communication and management, we chosen to abstract this away in our model, and instead model the latency / execution time overhead for the publish subscribe elsewhere in the model.

A RPC style interface can be modelled using the AADL `subprogram` access specification as shown in Listing 4.6

```
package planning
public
  with interfaces::planning;

  thread field_plan_generator extends
      interfaces::planning::plan_provider
    features
      get_plan : refined to provides subprogram access
        interfaces::planning::get_plan;
      get_planners : refined to provides subprogram access
        interfaces::planning::list_planners;
  end field_plan_generator;
end planning;
```

Listing 4.6: RPC style interface modelling

4.4 Modelling the ROS message passing infrastructure

The publish/subscribe message passing system used in ROS is based on TCP sockets. All nodes in a ROS system registers with a master who then provides the connection details when a node publishes/subscribes to a topic. The actual message passing is done directly between the nodes. The message passing system itself is transparent to the node, but we want to model the non-functional properties of the message passing system, in order to more accurately determine latencies associated with node to node communication.

In AADL we can model this as a `virtual bus` which can be used to model required protocols for communication. The model of the message passing system can be seen in Listing 4.7. The numbers used for the `Transmission_Time` are fictional, and will be estimated on real hardware platforms in section 5.2. In order to model that the data transmitted between a set of nodes is transported via the ROS message passing system, we can use the `Actual_Connection_Binding` property to bind the connection to the virtual bus.

```
virtual bus ros_ipc
  properties
    SEI::BandWidthCapacity => 10.0 GBytesps;
    Latency => 2 us .. 1 ms;
    Transmission_Time => [Fixed => 1 us .. 20 us;
      PerByte => 1 ns .. 1 ns;];
end ros_ipc;
```

Listing 4.7: virtual bus model of the ROS message passing system

4.5 Message type modelling

In order to be able to estimate timing properties of a system modelled in AADL the data types of the inputs and outputs must be known, or to be more specific, the expected size of the data transmitted between two components must be known. For the ROS middleware, we have developed a script which converts all defined message types into an corresponding AADL model. The ROS message types are comparable to C structures, they can either be a simple field like, `double,float,int,short,char`, an array or a reference to another defined structure. The conversion tool sums up the size of the individual fields recursively until the complete size is estimated. The complete size cannot be determined accurately since the size of the arrays is unspecified in the ROS message type definitions. Per default an array is assumed to be containing one element. However the modelling capabilities of AADL makes it possible to override the default size by assigning the property `Data_Size` another value. Listing 4.8 shows an example of a ROS message described in AADL.

```
data Imu
properties
    Data_Size => 128 Bytes;
end Imu;

data implementation Imu.impl
subcomponents
 linear_acceleration_covariance : data Base_Types::Float_64[];
 orientation : data middleware::ros::geometry_msgs::Quaternion.impl;
 angular_velocity_covariance : data Base_Types::Float_64[];
 orientation_covariance : data Base_Types::Float_64[];
 header : data middleware::ros::std_msgs::Header.impl;
 linear_acceleration : data middleware::ros::geometry_msgs::Vector3.impl;
 angular_velocity : data middleware::ros::geometry_msgs::Vector3.impl;
end Imu.impl;
```

Listing 4.8: AADL model of the sensor_msgs/Imu ROS message type

It is possible to associate units with the message type definitions, in order to make it clear the expected units that the sender and receiver of a given message should use.

4.6 Component parameters

AADL does not provide explicit support for modelling parameterisation of components. As an example consider an component converting general linear/angular velocity commands into left/right wheel velocities command. One important parameter the component need to know is the distance between the two wheels. This can be modelled implicitly in AADL by defining a `data` type and implementation containing these parameters as shown in Listing 4.9.

```
data gnss_localisation_config
end gnss_localisation_config;

data implementation gnss_localisation_config.impl
subcomponents
  orientation_buffer_size : data Base_Types::Integer {
    Data_Model::Integer_Range => 2 .. 50;};
  min_dist : data Base_Types::Float {
    Data_Model::Real_Range => 0.04 .. 1.5;
    Data_Model::Measurement_Unit => "m";};
  min_vel : data Base_Types::Float{
    Data_Model::Real_Range => 0.05 .. 1.0;
    Data_Model::Measurement_Unit => "m/s";};
end gnss_localisation_config.impl;
```

Listing 4.9: Sample of data type and unit specification for the parameters of a component

The properties associated with the data components in Listing 4.9 can be further detailed with default values (`Data_Model::Initial_Value`). The `Measurement_Unit` property used for representing units is based on text strings, which can result in problems with units being the same but written differently, for a more formal type based unit, a `property set` can be declared enumerating the units as proper AADL units to be applies to a data element.

5

Model validation and refinement

In this chapter some of the models developed in chapter 2, 3 and 4 are validated through experiments. Furthermore a method for obtaining the platform timing data needed for the analysis of the communication delays between software components is presented along with concrete measurements of different hardware platforms. Finally a method for measuring the execution time of software components is presented and concrete results for a selected set of components is presented. The models are validated on the hardware platforms listed in Table 5.1

Computer	Processor	N cores	Clock Freq	Memory	Disk type
RT101	i.MX6 A9	4	1.0 GHz	1 GB	Class 10, 8 GB SD card
Spectra	i7-3612QM	2	2.1GHz	8GB	Samsung SSD 840 256 GB

Table 5.1: Table of the hardware platforms used for validating the models

5.1 Validating bandwidth estimates

The hardware plays an important role when the latency of a system has to be determined. Communication busses interconnects the computer with physical devices.

5.1.1 Ethernet based devices

The experiments performed in this section aims to quantify the bandwidth required for communication from a serial to ethernet converter device. These experiments also provide a guideline for how to estimate the bandwidth of a ethernet based device.

The bandwidth requirements for a given ethernet device can be modelled in AADL and then a experiment can be carried out in order to validate this model against the real device. Figure 5.1 shows the AADL model of an IMU with a serial interface which is bridged to ethernet via an adapter. The purpose of the experiment is to compare the estimated bandwidth usage from the AADL model with the measured actual bandwidth. The bandwidth estimate from the AADL model is calculated by using the bandwidth estimation tool from OSATE as demonstrated in section 3.2. The bandwidth estimation is based on the size of the data transmitted from the serial device, the overhead added by the TCP protocol when transmitting the data via ethernet and finally the output rate of the IMU. The data sent from the IMU varies in size due to numbers being

transmitted as ASCII, and this is difficult to include in the bandwidth model. Therefore we use a "guesstimated" mean data size for the model.

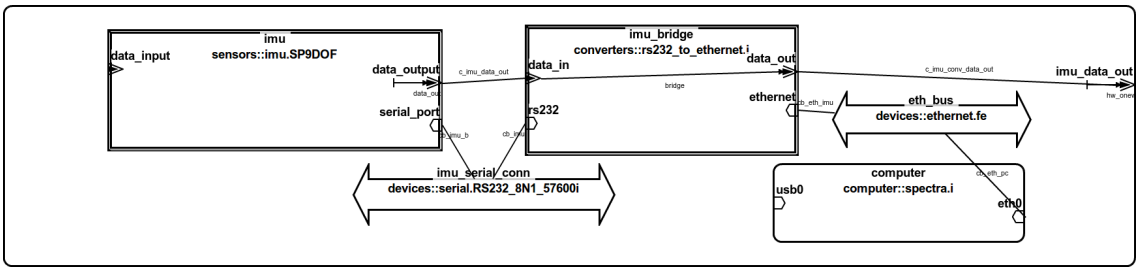


Figure 5.1: AADL model of the ethernet bridge between a serial device and a computer.

Wireshark[1] is used for measuring the bandwidth required by the setup shown in Figure 5.1. Each packet received from the device to the computer is stored with a timestamp of reception. The total bandwidth usage is then calculated as

$$BW_{total} = \frac{\sum_{p=0}^{N} packets(p)}{t_{end} - t_{start}} \tag{5.1}$$

Where p is the packet number, $packets$ is a function returning the size of that package in bytes, and t_{start} is the time of the first package, and t_{end} is the time in seconds for the last package.

	Total packet size [Bytes]	**Total time [s]**	**Bandwidth [Bytes/s]**
Estimated	na	na	4560
Measured	88156	18.048	4884.5

Table 5.2: Estimated bandwidth usage from the AADL model and actual measured bandwidth usage.

Table 5.2 shows the estimated bandwidth of the ethernet connection between the IMU bridge and the computer and the actual measured bandwidth. The estimated bandwidth is lower than the actual measured. This may be due to the data size variation. The relative error between the estimated bandwidth and the actual measured is 6.64%.

5.2 Profiling communication delay of the ROS middleware

In order to determine the communication delay of the ROS publish/subscribe middleware on a given hardware platform, a set of experiments has been designed. The experiments are divided into two parts, one part for local communication on the same computer between two ROS nodes (processes), and another part for communication between two nodes on separate computers connected via ethernet. An overview of the experiment setup for the local communication can be seen in Figure 5.2 and the setup for the remote communication can be seen in Figure 5.3.

From Figure 5.4 we can see that the standard deviation of the transmission delay between two ROS nodes running on the same computer is depending on the message size itself. This dependency is difficult to model in AADL at least using only a `virtual bus` as the representation of the middleware. However we can decide on suitable linear values for a given situation, and thereby avoid complex modelling of the middleware. Furthermore the data presented in Figure 5.4 shows

[1] www.wireshark.org

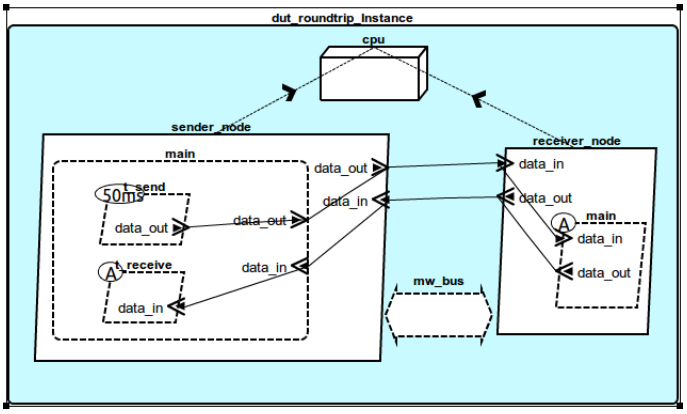


Figure 5.2: AADL Model of the experiment setup for local node communication measurements, where both nodes are located on the same computer.

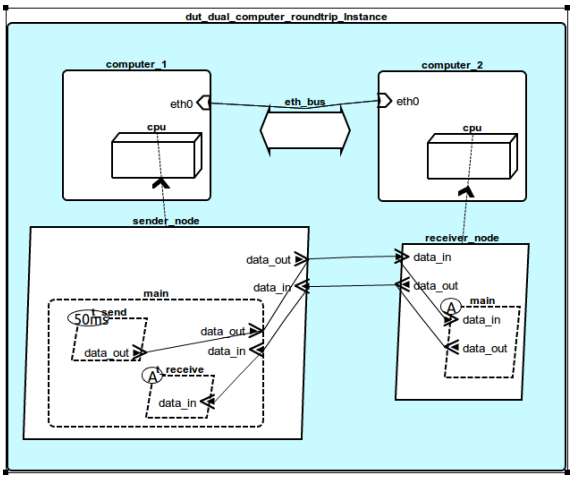


Figure 5.3: AADL Model of the experiment setup for remote node communication measurements where the sender is located on the Spectra PC and the receiver is located on RT101 embedded platform, connected via a 1000 Mbit/s ethernet.

a few outliers. These outliers may come from the Linux operating system it self, as the kernel is not real-time capable, and therefore does not provide any deadline guarantees. section 5.4 will provide more details on the scheduling of a ROS node.

Data size	Computer	min [ms]	max [ms]	mean [ms]	median [ms]	stddev [ms]
76 Bytes	RT101	1.0	2.5	1.2	1.2	0.056
	Spectra	0.206	0.555	0.468	0.478	0.049
132 Bytes	RT101	1.1	2.1	1.2	1.2	0.057
	Spectra	0.227	0.593	0.495	0.508	0.048
244 Bytes	RT101	1.1	2.2	1.3	1.3	0.075
	Spectra	0.218	0.567	0.494	0.510	0.043
468 Bytes	RT101	1.2	2.1	1.3	1.3	0.059
	Spectra	0.252	0.566	0.513	0.518	0.025
916 Bytes	RT101	1.2	2.3	1.3	1.3	0.056
	Spectra	0.234	0.607	0.486	0.499	0.048
1812 Bytes	RT101	1.3	2.8	1.5	1.4	0.096
	Spectra	0.254	0.671	0.487	0.497	0.044
3604 Bytes	RT101	1.7	2.8	1.8	1.8	0.095
	Spectra	0.306	0.687	0.547	0.551	0.026
7188 Bytes	RT101	2.3	3.8	2.4	2.4	0.077
	Spectra	0.300	0.731	0.594	0.600	0.034
14356 Bytes	RT101	3.6	5.0	3.7	3.7	0.093
	Spectra	0.429	0.859	0.670	0.676	0.046
28692 Bytes	RT101	6.1	7.6	6.2	6.2	0.070
	Spectra	0.669	1.2	0.918	0.923	0.036
57364 Bytes	RT101	11.1	13.2	11.2	11.2	0.133
	Spectra	1.0	1.7	1.4	1.4	0.058

Table 5.3: Combined results of each measured data size for the RT101 platform and the Spectra platform for easier comparison. Each value in the rows are calculated based on 2000 samples.

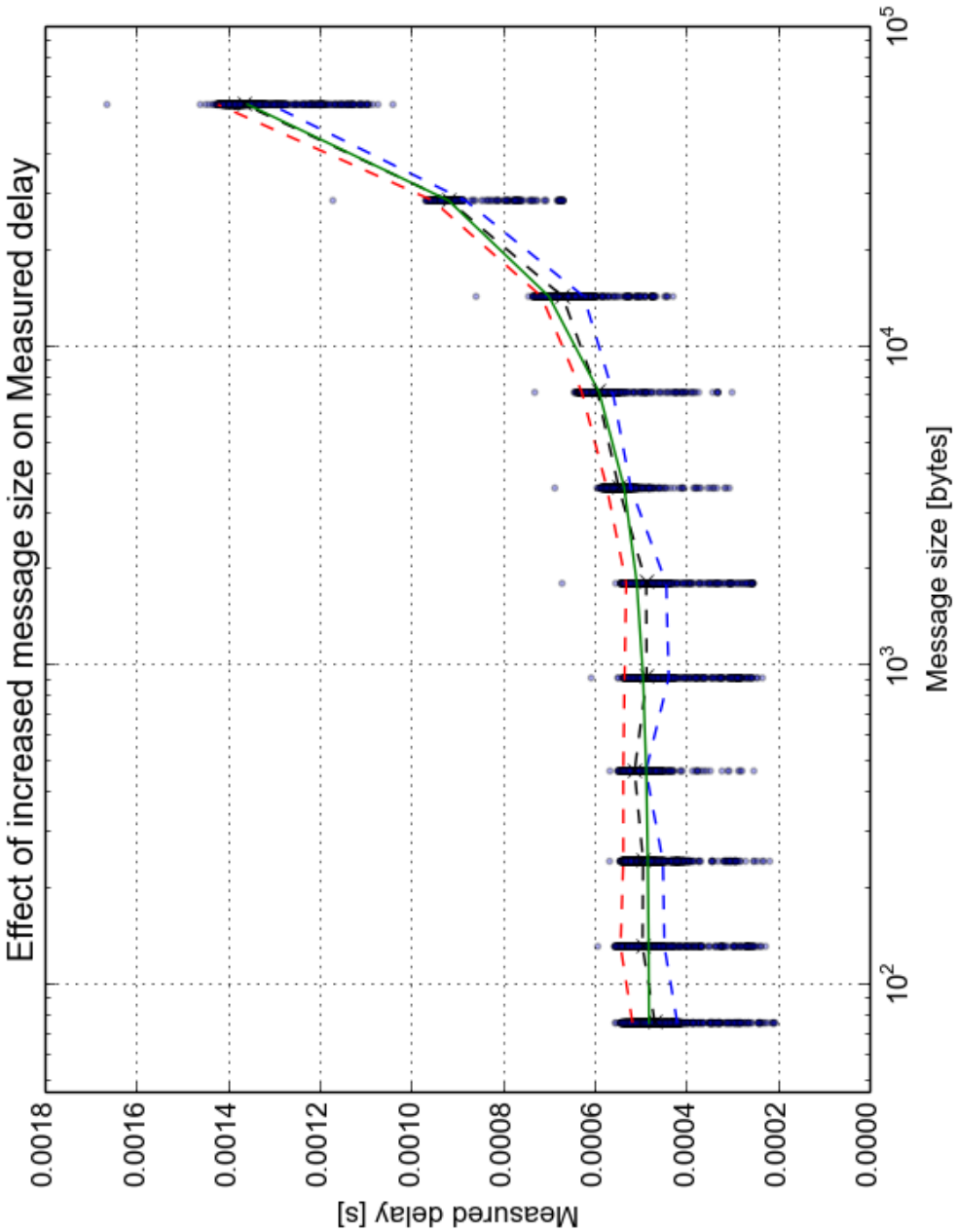


Figure 5.4: Result of running the roundtrip test on the spectra PC, each data size was measured 2000 times. The dashed black line represents the mean value, the dashed red line is the mean value plus the standard deviation, the dashed blue line is the mean value subtracted the standard deviation, the green line represents a linear regression on the mean values (the dashed black line), finally each sample is plotted as a dot.

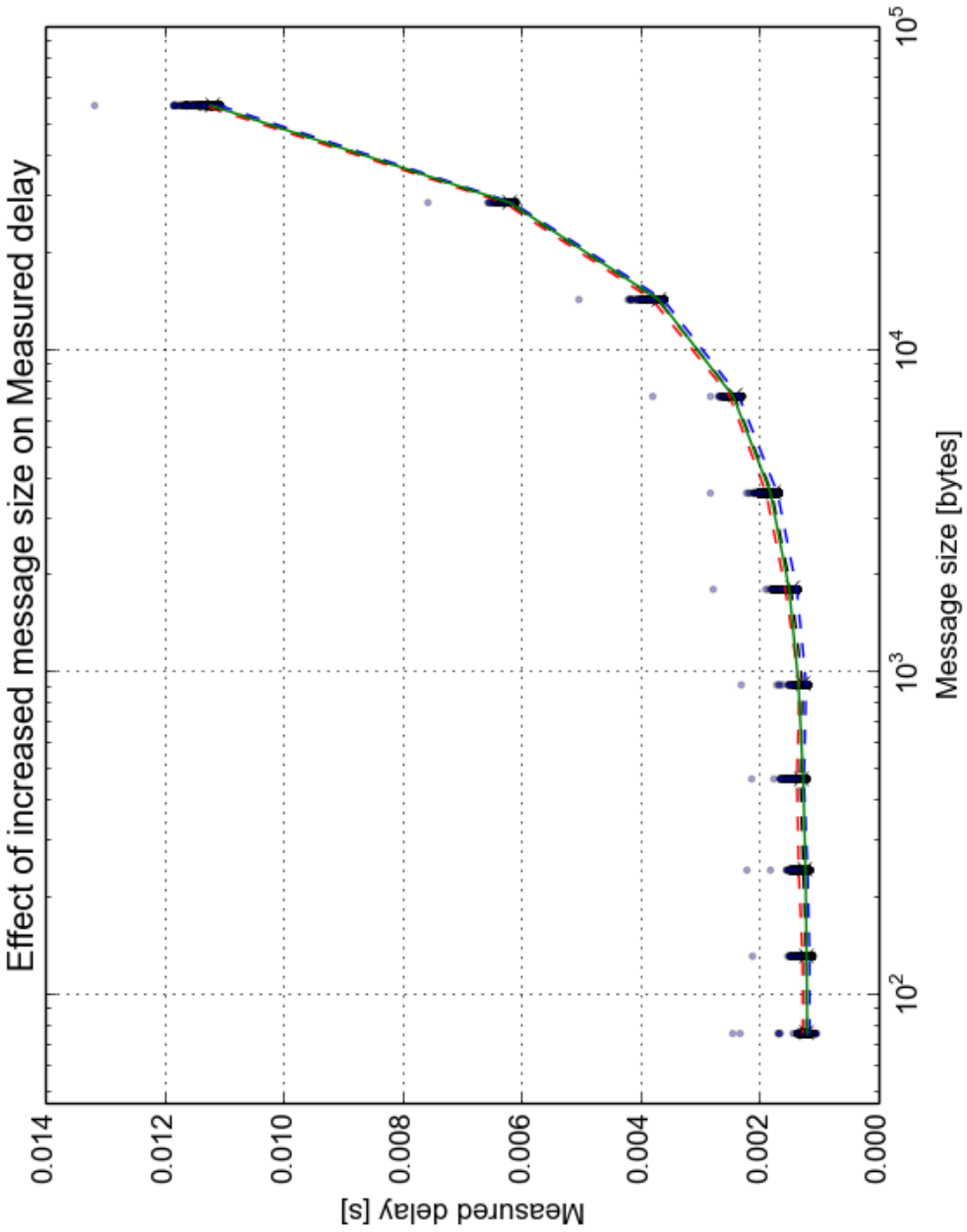


Figure 5.5: Result of running the roundtrip test on the rt101 embedded platform, each data size was measured 2000 times. The dashed black line represents the mean value, the dashed red line is the mean value plus the standard deviation, the dashed blue line is the mean value subtracted the standard deviation, the green line represents a linear regression on the mean values (the dashed black line), finally each sample is plotted as a dot.

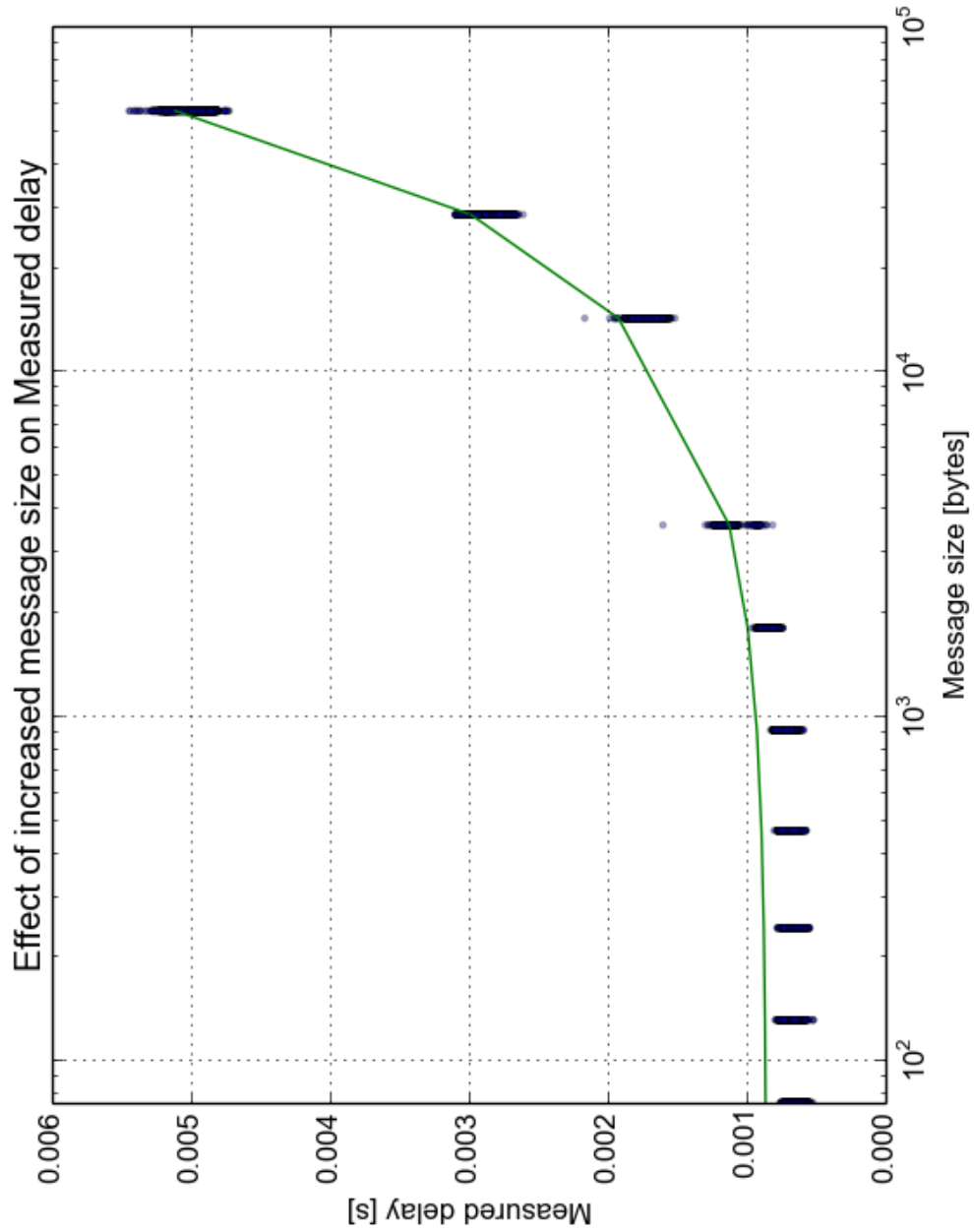


Figure 5.6: Result of running the roundtrip test with the sender node on the Spectra computer and the receiver running on the RT101 computer, each data size was measured 2000 times. The green line represents a linear regression on the mean values (the dashed black line), finally each sample is plotted as a dot. This plot of the measurement data excludes a number of outliers, and furthermore excludes the measurements for the 7188 Bytes data size, see section 5.3.

5.3 Network measurement anomalies

The networked node experiment performed in section 5.2 contained results for a specific data size, which varied in measured delay significantly more than the measured delay for the other data sizes. This section investigates this anomaly in more detail. In order to figure out when this behaviour starts to show in the measurements, the same test as performed in section 5.2 with a linear sweep from 76 Bytes up to 14020 Bytes with an increment of 560 Bytes. Figure 5.7 shows a scatter plot of these measurements.

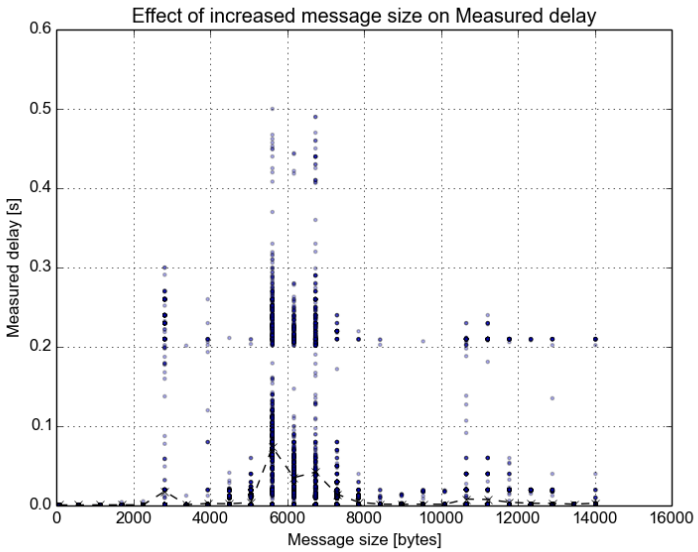
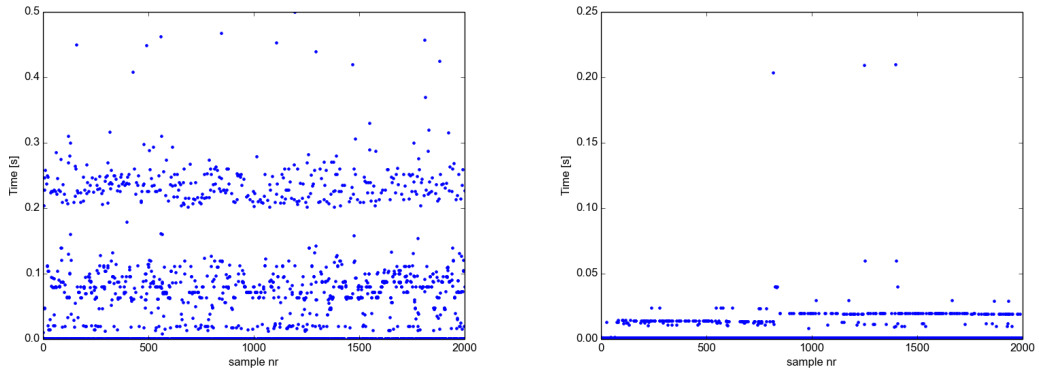


Figure 5.7: Scatter plot of the sweep from 76 Bytes to 14020 Bytes, with 560 Bytes increment. The round trip delay time was measured 2000 times for each message size. The dashed black line represents the mean value of each data size.

As seen from Figure 5.7, the variance starts to increase at around 3000 Bytes and then increases significantly at 5500 to around 7500 Bytes.

Figure 5.8 shows a comparison of two data sizes close to each other in data size, but with different measurement results. The root cause of this sudden increase in variance has not been found, but using Wireshark, a large amount of TCP retransmission frames where recorded for the measurements with high latency, although this is just a symptom of ethernet frames being dropped by the kernel.



(a) (b)

Figure 5.8: Plot of the 2000 measurements of the round trip delay with a data size of 5620 Bytes (a) and data size 5060 (b), showing a large variation in the delay measured in (a) and a much smaller variation in (b).

5.4 Profiling the Execution time of ROS nodes

In this section a method for measuring the execution time of a ROS component and using the execution time measurements in an AADL setting is presented. Table 5.4 list the components to be profiled. The listed components are all part of the flow from the GNSS device to the sprayer device.

Component	Type
gnss_serial_driver	Asynchronous
gnss_pose_provider	Asynchronous
ekf_pose_provider	Asynchronous
auto_steering	Periodic
implement_controller	Hybrid
implement_driver	Asynchronous

Table 5.4: A list of the components and their implementation pattern, for which the execution time is measured and feed back into the AADL model for further analysis

Each component is measured independently of the other, in a component specific test bench which provides sample inputs for the component. The component is instrumented with execution time measurement code. For components implemented as a periodic node, the execution time of the timer loop is measured. For asynchronous nodes, each callback is measured. For hybrid nodes, both the timer loop and the callbacks are measured.

Component	Thread	Platform	Min	Max	Mean	Stddev
gpgga_to_utm	gpsStateCallback	rt101	23.7us	137.0us	30.5us	3.2us
		spectra	4.9us	48.1us	20.2us	2.7us
gnss_pose_provider	ongnss_subMsg	rt101	243.0us	1.3ms	432.0us	205.5us
		spectra	73.2us	756.8us	165.3us	33.5us
ekf_pose_provider	onOdomMessage	rt101	7.3us	1.6ms	456.9us	162.0us
		spectra	1.2us	233.5us	132.1us	45.9us
	onTimer	rt101	1.7us	2.8ms	53.1us	53.6us
		spectra	486.0ns	1.6ms	41.4us	31.0us
	onUtmMessage	rt101	8.0us	1.9ms	868.8us	306.1us
		spectra	1.2us	1.2ms	215.1us	77.4us
auto_steering	on_timer	rt101	4.3us	3.4ms	85.8us	112.7us
		spectra	1.1us	631.9us	48.2us	25.1us
	onpath_cmd_inMsg	rt101	3.3us	16.9ms	32.7us	694.9us
		spectra	1.6us	3.1ms	8.7us	125.5us
implement_controller	on_timer	rt101	13.7us	83.7us	28.0us	4.3us
		spectra	6.9us	63.3us	21.0us	2.2us
	oncmd_inMsg	rt101	698.7us	698.7us	698.7us	0.0
		spectra	226.8us	226.8us	226.8us	0.0
	onpath_feedbackMsg	rt101	22.3us	507.0us	30.1us	13.0us
		spectra	4.4us	224.2us	17.3us	6.0us

Table 5.5: Table showing the min,max,mean and standard deviation of the computation time for each thread in each component on the two hardware platforms. The components are part of the GNSS device to sprayer flow path.

5.5 Refining the system model

In this section we update the AADL model of the Intelligent Marking Robot with the timing measurement results obtained from the two computer platforms in section 5.4 and section 5.2.

5.5.1 Determine model parameters

The first task is to derive the model parameters for the middleware from the experimental data obtained in section 5.2. In our case we are interested in the variation in the latency.Therefore we want to extract a reasonable estimate for the minimum transmission time and the maximum transmission time. The minimum transmission time is calculated based on fitting a linear function to the minimum transmission value of each measured data size, and then divide this by two as the results in section 5.2 are round trips.

Platform	Min per-byte	Min fixed	max per-byte	Max fixed
RT101	87 ns	529 us	95 ns	1115 us
Spectra	7 ns	113 us	10 ns	295 us

Table 5.6: The linear function parameters for the AADL bus transmission time property - calculated from the measurement results obtained in section 5.2, Table 5.3, by fitting a linear function to the Min value of each data size, and fitting a linear function to the Max value of each data size, and then divide by two to account for the measurements being roundtrip.

The choice of model parameters differs based on the system to be analysed and the type of analysis to be performed, in our case we are interested in a worst case scenario when it comes to the maximum delay and minimum delay, but other model parameters may be more suitable for analysing the generic case of latency and execution time, if the system requirements are different. For each of the threads involved in the flow path from the GNSS device to the sprayer device, we use the minimum and maximum observed execution time from Table 5.5 as model parameters for the `Timing_Properties::Compute_Execution_Time`.

5.5.2 Latency analysis

The flow latency from the GNSS device to the sprayer device is now analysed again using the model of the robot refined with the execution time measurements and the middleware overhead measurements. The flow analysed includes only the software components, and the minimum and maximum reported delays or executions times are used. Table 5.7 shows the latency contribution by each thread and connection between ports of these threads. The main contribution to the latency variance comes from the sampling of the data at the `ekf_fusion.main_loop` thread and at the `nav_controller` thread. The total variation in latency for the RT101 platform is $43ms$ which results in a $2.5cm$ variation in the resulting start and end of a painted line. The line width is commonly $10cm$ and if the route is generated such that the sprayer targets to turn on and off at the center then this variation is within the requirements. However, this variation is only part of the total variation of the line. The accuracy of the GNSS and the navigation controllers also affects the total precision of the sprayed line.

Table 5.9 shows the latency contribution for each thread and connection when executing on the Spectra computer, a more powerful platform. As seen we do not get a large reduction in the latency variation just be selecting a more powerful platform, as the main contribution to the latency variation is the periodic sampling of the inputs by the `ekf_fusion.main_loop` thread and at the `nav_controller` thread.

Contributor	Min Value	Min Method	Max Value	Max Method
Connection	0.53161ms	no latency	1.11785ms	no latency
thread gnss	0.0ms	queued	0.0ms	queued
thread gnss	0.023ms	processing time	0.0ms	no latency
Connection	0.5377ms	no latency	1.1245ms	no latency
thread gnss_provider	0.0ms	queued	0.0ms	queued
thread gnss_provider	0.243ms	processing time	0.0ms	no latency
Connection	0.542224ms	no latency	1.12944ms	no latency
thread ekf_fusion.gnss_filter	0.0ms	queued	0.0ms	queued
thread ekf_fusion.gnss_filter	0.008ms	processing time	0.0ms	no latency
Connection	0.0ms	no latency	0.0ms	no latency
thread ekf_fusion.main_loop	0.0ms	sampling	20.0ms	sampling
thread ekf_fusion.main_loop	0.002ms	processing time	0.0ms	no latency
Connection	0.542224ms	no latency	1.12944ms	no latency
thread nav_controller	0.0ms	sampling	20.0ms	sampling
thread nav_controller	0.0ms	queued	0.0ms	queued
thread nav_controller	0.004ms	processing time	0.0ms	no latency
Connection	0.5608ms	no latency	1.14977ms	no latency
thread implement_controller.async	0.0ms	queued	0.0ms	queued
thread implement_controller.async	0.014ms	processing time	0.0ms	no latency
Connection	0.529087ms	no latency	1.115095ms	no latency
thread sprayer	0.0ms	queued	0.0ms	queued
thread sprayer	0.0ms	no latency	0.0ms	no latency
Connection	0.53ms	no latency	1.11614ms	no latency
Latency Total	4.07ms		47.88ms	

Table 5.7: OSATE Flow Latency report for the refined instance of the Intelligent Marking model using the RT101 computer platform, the hardware and low-level drivers are excluded for this analysis.

Table 5.8

Contributor	Min Value	Min Method	Max Value	Max Method
Connection	0.113ms	no latency	0.295ms	no latency
thread gnss	0.0ms	queued	0.0ms	queued
thread gnss	0.005ms	processing time	0.0ms	no latency
Connection	0.114ms	no latency	0.296ms	no latency
thread gnss_provider	0.0ms	queued	0.0ms	queued
thread gnss_provider	0.073ms	processing time	0.0ms	no latency
Connection	0.114ms	no latency	0.296ms	no latency
thread ekf_fusion.gnss_filter	0.0ms	queued	0.0ms	queued
thread ekf_fusion.gnss_filter	0.001ms	processing time	0.0ms	no latency
Connection	0.0ms	no latency	0.0ms	no latency
thread ekf_fusion.main_loop	0.0ms	sampling	20.0ms	sampling
thread ekf_fusion.main_loop	4.86E-4ms	processing time	0.0ms	no latency
Connection	0.114ms	no latency	0.296ms	no latency
thread nav_controller	0.0ms	sampling	20.0ms	sampling
thread nav_controller	0.0ms	queued	0.0ms	queued
thread nav_controller	0.001ms	processing time	0.0ms	no latency
Connection	0.115ms	no latency	0.299ms	no latency
thread implement_controller.async	0.0ms	queued	0.0ms	queued
thread implement_controller.async	0.007ms	processing time	0.0ms	no latency
Connection	0.113ms	no latency	0.295ms	no latency
thread sprayer	0.0ms	queued	0.0ms	queued
thread sprayer	0.0ms	no latency	0.0ms	no latency
Connection	0.113 ms	no latency	0.295 ms	no latency
Latency Total	0.88 ms		42.07 ms	

Table 5.9: OSATE Flow Latency report for the refined instance of the Intelligent Marking model using the Spectra computer platform, the hardware and low-level drivers are excluded for this analysis.

6

Evaluation

This this chapter summaries the contents of the previous chapter and provides concluding remarks.

6.1 Concluding remarks

From a system and architecture design point of view, the AADL language is easily applied, and can be used to structure a design into both hardware and software sub-systems for which the interfaces can be defined, and refined. The hardware modelling aspects of AADL enable early analysis and selection of Bus bandwidths, communication protocols, etc. Furthermore being able to experiment with different architectures at a relative low effort, is paramount for achieving an architecture which satisfied all the requirements. From a software modelling point of view, AADL is most suited for modelling components as black-boxes in terms of behaviour, and then focusing on the interface and execution semantics (asynchronous, periodic, etc.). The Flow Path Analysis plugin offers key insight into the design tradeoffs, by providing detailed impact of assigning a thread a specific period, in terms of overall system performance. However the absolute accuracy of the predictions has not been validated in this work. However as a initial comparison between different architectures it is still a valuable tool.

Modelling the robotic middleware as a virtual bus, enables different middlewares to be benchmarked in the same way as the ROS middleware was, and compare the gain from switching middleware, or to model subsystems using a different middleware for communication. As seen from the middleware measurements, the ROS middleware has quite varying performance, in terms of message transmission delay. The reason for the measured delays, has not been investigated, but a primary influence on the results, is the fact that the system is not real-time. Many robotic systems contains both real-time tasks and non-real-time tasks. Real-time tasks may be low-level motor control running at 500Hz, or functions which have a hard deadline in terms of reaction time. Tasks such as feature extraction, and Simultaneous Localisation and Mapping (SLAM) algorithms typically does not execute in a real-time environment due to their unpredictability in terms of execution time, and the amount of computational resources required. However AADL can handle both types of systems, although it is difficult to express that the scheduler is not real-time, as this does not affect any of the end-to-end analyses made. But from a modelling point of view, it is trivial to have multiple systems running on different processors, and using different schedulers in the same overall model.

Bibliography

[1] Biggs, G., Fujiwara, K., Anada, K.: Modelling and analysis of a redundant mobile robot architecture using aadl. In: Simulation, Modeling, and Programming for Autonomous Robots, pp. 146–157. Springer (2014)

[2] Bruyninckx, H.: Open robot control software: the orocos project. In: IEEE ICRA 2001 Proceedings. vol. 3, pp. 2523–2528 vol.3 (2001)

[3] Delange, J., Feiler, P.: Incremental latency analysis of heterogeneous cyber-physical systems (2014)

[4] Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)

[5] Fernández, M.M.: Ground systems modeling using the architecture analysis & design language (aadl) (2014)

[6] Jang, C., Lee, S.I., Jung, S.W., Song, B., Kim, R., Kim, S., Lee, C.H.: Opros: A new component-based robot software platform. ETRI journal 32(5), 646–656 (2010)

[7] Jensen, K., Larsen, M., Nielsen, S.H., Larsen, L.B., Olsen, K.S., Jørgensen, R.N.: Towards an open software platform for field robots in precision agriculture. Robotics 3(2), 207–234 (2014)

[8] Jørgensen, R., Sørensen, C., Maagaard, J., Havn, I., Jensen, K., Søgaard, H., Sørensen, L.: Hortibot: A system design of a robotic tool carrier for high-tech plant nursing. Agricultural Engineering International: CIGR Journal (2007)

[9] Linz, A., Ruckelshausen, A., Wunder, E., Hertzberg, J.: Autonomous service robots for orchards and vineyards: 3d simulation environment of multi sensor-based navigation and applications (2014), https://my.hs-osnabrueck.de/ecs/fileadmin/groups/156/Veroeffentlichungen/2014-ICPA_2014_Autonomous_Service_Robots_for_Orchards_and_Vineyards_3D_Simulation_Environment_of_Multi_Sensor_Based_Navigation_and_Applications.pdf

[10] Murugesan, A., Heimdahl, M., Whalen, M., Rayadurgam, S., Komp, J., Duan, L., Kim, B., Sokolsky, O., Lee, I.: From requirements to code: Model based development of a medical cyber physical system (2014)

[11] Nebot, P., Torres-Sospedra, J., Martínez, R.J.: A new hla-based distributed control architecture for agricultural teams of robots in hybrid applications with real and simulated devices or environments. Sensors 11(4), 4385–4400 (2011)

[12] Noll, T.: Safety, dependability and performance analysis of aerospace systems. In: Formal Techniques for Safety-Critical Systems, pp. 17–31. Springer (2014)

[13] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. No. 3.2 (2009)

[14] SAE: Architecture Analysis & Design Language (AADL). SAE, SAE (September 2012), http://standards.sae.org/as5506b/

[15] Schlegel, C., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven software development. INTECH Open Access Publisher (2012)

[16] Senn, E., Laurent, J., Diguet, J.P.: Multi-level power consumption modelling in the aadl design flow for dsp, gpp, and fpga. ACESMB 2008 p. 9

[17] Yu, H., Yang, Y.: Latency analysis of automobile abs based on aadl. In: Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on. pp. 1835–1838. IEEE (2012)

Morten Larsen, Modelling field robot software using AADL, 2016

Department of Engineering
Aarhus University
Inge Lehmanns Gade 10
8000 Aarhus
Denmark

Tel.: +45 8715 0000