MASTER THESIS

Model-checked
Space Plug-and-Play Architecture
Local Subnet Adaptation
implemented in Ada with
Ravenscar restrictions

Author
Christoffer Holmstedt

Supervisor
Adj. Prof. Fredrik Bruhn, PhD

Examiner
Prof. Kristina Lundqvist

MÄLARDALEN UNIVERSITY
School of Innovation, Design and Engineering

November 2, 2014
Abstract

Space Plug-and-Play Architecture (SPA) is a set of standards to make it easier to build small satellites. Focus is put on improving the integration phase and the time consuming validation and verification process by introducing plug-and-play functionality. From mission call-up to operational satellite it should only take six days.

A SPA network consists of several different types of subnets with different pros and cons. For each processing node there must be one Local Subnet Manager (SM-L). The SM-L can communicate over different communication protocols depending on how the respective local subnet is set up, one option is UDP/IP.

In this thesis Ada Protected Objects is presented as a viable option for inter-process communication instead of UDP/IP in a SPA network. This thesis presents the initial work towards a SPA Local Subnet Adaptation that builds on language constructs in Ada such as Ada Tasks and Protected Objects. The system design and implementation is verified deadlock free with UPPAAL but shows indications of livelock possibilities. The severity of these livelock situations is discussed in the conclusion.

Keywords: Space plug-and-play Architecture, SPA, UPPAAL, Timed automata, Ada, Ravenscar, Unified Common Architecture, UNICA, Virtual Network Protocol, VNP
Contents

List of Abbreviations iii

1 Introduction 1
  1.1 Problem Statement ........................................... 2
  1.2 Structure of this report ..................................... 3

2 State of the Art 4
  2.1 Space plug-and-play Architecture .......................... 4
  2.2 UPPAAL .......................................................... 6
  2.3 Internet of Things ............................................. 7

3 Method 9
  3.1 Literature study ............................................... 9
  3.2 An Agile Unified Process .................................... 10

4 Space Plug-and-Play Architecture 11

5 Designing and modeling the system 14
  5.1 First iteration ............................................... 15
  5.2 Second iteration ............................................. 18
  5.3 Third iteration .............................................. 22
  5.4 Verification .................................................. 23

6 Result 28
  6.1 Model verification ........................................... 28
  6.2 Implementation .............................................. 28

7 Conclusion 31

Bibliography 37

A System Design 38
  A.1 Virtual Network Library implementation, high-level design . 38
  A.2 Component and Service discovery process ................... 39
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFRL</td>
<td>Air Force Research Laboratory</td>
</tr>
<tr>
<td>AIAA</td>
<td>American Institute of Aeronautics and Astronautics</td>
</tr>
<tr>
<td>BAP</td>
<td>Bruhnspace Advanced Projects AB</td>
</tr>
<tr>
<td>CAS</td>
<td>SPA Central Addressing Service</td>
</tr>
<tr>
<td>CDD</td>
<td>Common Data Dictionary</td>
</tr>
<tr>
<td>CUUID</td>
<td>Component Universally Unique Identifier</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LS</td>
<td>SPA Lookup Service</td>
</tr>
<tr>
<td>MDH</td>
<td>Mälardalen University</td>
</tr>
<tr>
<td>SDM</td>
<td>System Data Model</td>
</tr>
<tr>
<td>SM-L</td>
<td>SPA Local Subnet Manager</td>
</tr>
<tr>
<td>SPA</td>
<td>Space Plug-and-Play Architecture</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UNICA</td>
<td>Unified Common Architecture</td>
</tr>
<tr>
<td>VNP</td>
<td>Virtual Network Protocol</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
<tr>
<td>xTEDS</td>
<td>Extensible Transducer Electronic Data Sheet</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Space plug-and-play Architecture (SPA) is intended to decrease the time needed in the integration phase when constructing satellites for specific missions. This is done by introducing plug-and-play interfaces between different components, both software and hardware components. SPA is a set of standards published by "American Institute of Aeronautics and Astronautics" (AIAA). The different standard documents define everything from power supply interfaces to how components automatically describe themselves to other components.

SPA has been developed since 2004 and has been shown to work as intended. As of early 2014 efforts are now taking place to broaden the scope of SPA under the names of ”Unified Common Architecture” (UNICA) and ”Virtual Network Protocol” (VNP). The goal with Unified Common Architecture is to introduce SPA concepts and ideas to the field of robotics, unmanned vehicles, industrial automation and internet of things. This work is done in a collaboration between Mälardalen University (MDH) and Bruhnspace Advanced Projects AB (BAP) in Sweden.

VNP is a subset of the SPA standards that strictly limit the focus to the software parts of SPA. The goal is to try to stay interoperable with SPA as much as possible though some differences might be necessary to introduce.

As a part of SPA is the ”inter-process communication” (IPC) through UDP/IP within respective processing node. The dependability of UDP/IP can be discussed but the starting point for this thesis work is that UDP/IP is not good enough, better options exist.

To increase dependability of a SPA network this thesis work will look at the possibilities of developing core parts of SPA in the Ada programming language which has a good track record in the field of safety-critical systems.
Focus will be on the local subnet which exists on each processing node. During implementation the system will be modeled with UPPAAL to verify specified requirements.

Before describing the problem statement it is important to define a few key terms. Both the term safety-critical system and high-integrity system are commonly used in different fields of research and communities. In this report the term safety-critical system will be used and the definition of a safety-critical system is that it is dependable. "Dependability" is a broad term and in this thesis the definition used by Avižienis et al. [5] and Laprie [16] is used. More specifically the attributes in focus are availability, reliability, safety, integrity, maintainability and resilience in no specific order. Confidentiality is deemed out of scope and left for future work.

1.1 Problem Statement

This thesis work will focus on the core parts of a SPA network which runs on each processing node. This includes the Central Addressing Service (CAS), the Lookup Service (LS) and the Subnet Manager for "Local Subnets" (SM-L) components. Only one instance of the CAS and LS runs within each SPA network at the same time but each processing node must have a SM-L running. Each component will be implemented as one or several Ada Tasks and communication between the Ada Tasks will be through Ada Protected Objects which is a shared memory construct defined as a part of the Ada programming language.

For SPA interoperability the SM-L will have two interfaces to other SPA components one to other local Ada Tasks through dedicated protected objects and one through UDP/IP. The interface to external systems, the UDP/IP interface, will be the system boundary of the analyzed system in this thesis.

The following list includes the questions to be answered

1. Does the system have any possibility to reach a state of deadlock?
2. Does the system have any possibility to reach a state of livelock?

To answer these questions focus was first put on reading literature on how to develop safety-critical systems in Ada. Designing the system, implementing in Ada and modeling in UPPAAL was done over a few iterations to reach the end result.
1.2 Structure of this report

The report is structured as follows. Chapter 2 goes through the state of the art of Space plug-and-play Architecture, UPPAAL modeling and verification as well as Internet of Things. Chapter 3 continues with the research and development method used throughout the thesis work, from literature study to modeling and implementation. A thorough presentation on how Space plug-and-play Architecture works is then given in chapter 4. Chapter 5 presents how the system design and UPPAAL model have evolved over the development iterations. Chapter 6 shows all the results acquired and chapter 7 finishes the report with the discussion about the results and suggestions for future research and development.
Chapter 2

State of the Art

This chapter on the state of the art in related research fields aims at giving the reader a basic stepping stone for later chapters where more details are described concerning Space plug-and-play Architecture and UPPAAL modeling.

2.1 Space plug-and-play Architecture

The Air Force Research Laboratory (AFRL) Space Vehicle Directorate started a project referred to as "Space plug-and-play Avionics" in 2004 with the intent to understand why space systems used in various missions became so complex and how they can be made easier to assemble [11, 26]. The solution was to look at plug-and-play systems already available on the ground. An example to this is how USB devices work with laptop and desktop computers in everyday life. It does not matter which keyboard that is used, basic functionality will work in any computer it is used with, no matter if it runs Windows, OS X or a Linux distribution.

"Space plug-and-play Avionics" was released during the 2000-decade as a set of standards and in an updated version in 2011 renamed to "Space plug-and-play Architecture" [26], all standards are available from AIAA [35].

Development and testing of implementations conforming to the standards have been ongoing since the launch of the program. One project that has been presented is the PnPSat by Fronterhouse et al. [11] and Martin et al. [25]. They have demonstrated the first created plug-and-play satellite and given their view on the problems encountered during development of both hardware and software components. The first plug-and-play satellite flown on orbit is presented by Martin et al. [26] where most parts of the
plug-and-play experiment passed all tests one year after launch.

As a critical part of plug-and-play satellite research and development is the "System Data Model" (SDM) middleware software presented by Sundberg et al. [38], previously under the name "Satellite Data Model" [23]. The SDM can be viewed in two different ways, as a middleware or as a "sideware" [11]. The SDM is one implementation of the SPA standards that manage the parts of network and data capabilities discovery. The SDM source code has been released under public domain license [23].

As this thesis work focus on Ada the SDM source code available in C++ could not be used but it relates closely. This thesis is the starting ground for an Ada implementation that would be interoperable with the SDM. In other words, a SPA network could consist of multiple SDM processing nodes and multiple processing nodes running the Ada source code.

To enable components to be self-describing all SPA components must have an "Extensible Transducer Electronic Data Sheet" (xTEDS) file stored within the component. The xTEDS file includes all the information other components need to understand its capabilities e.g. which interfaces the component has. The problem at this point is that multiple vendors start to create their own interface definitions and all benefits with self-describing components are lost. This is where the "Common Data Dictionary" (CDD) comes into play.

The first work to create the CDD is described by Pasko as an informal approach. Instead Pasko suggests a more formal approach using Resource Description Framework (RDF). Pasko suggests that RDF and RDF Schemas can be used to not only describe the CDD formally but also be used to visually represent the CDD with suitable software [31].

Hansen et al. [12] describes the dictionary as an ontology which consists of several taxonomies in a hierarchical structure. This means that if different vendors create SPA components with temperature sensors, the vendors use the CDD to describe the interfaces with the correct vocabulary. To manage the risk of adding too many terms in the CDD Lanza et al. [15] suggests the forming of a CDD panel to "[...] serve as a regulatory structure for the CDD". To make it easier for developers to write xTEDS files with the correct vocabulary and improve the CDD, Lanza et al. [15] have created a web based tool to manage the CDD.

One benefit of using XML file structure for the xTEDS is that a XML schema definition (XSD) can be created and with the help of a XML validating parser xTEDS can be verified automatically to conform to the xTEDS file format and the CDD [15].
2.2 UPPAAL

To verify that a system meets safety-critical standards different approaches can be taken and depending on the severity of failures that can be caused by the system different approaches must be taken. To verify that the core SPA system implementation presented in this paper meets some basic requirements it will be modeled and verified with the model-checking tool UPPAAL.

On the official website UPPAAL is described as follows, "UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)." [39]. It is maintained and developed by participants mainly from Uppsala University in Sweden and Aalborg University in Denmark.

The formal methods behind UPPAAL builds on the research originally presented by Alur et al. [4] where real-time systems are shown to be modeled with timed finite automata, or just timed automata in short. The key part is that the system has a finite amount of states so it is possible to model it in a network of automata. Since the presentation of the formal method, tools have been developed to make the method useful. UPPAAL has over the last decade been forked to more specific versions such as "TIMES" which can be used for schedulability analysis and "TRON" for online real-time testing, other forks exist and for a more detailed history of the development process "Developing UPPAAL over 15 years" by Behrmann et al. [7] is suggested. In this thesis "classic" UPPAAL is used to verify real-time properties of the system.

UPPAAL is used mainly to verify real-time systems in a wide variety of research fields. From the field of radiation therapy [24] and safety-critical systems in the nuclear engineering domain [14] to embedded system in a university satellite [1] and flight-program in an unmanned helicopter [18]. Worth noting is that UPPAAL can also be used to verify protocols such as in a distributed lift system with a CAN Bus presented by Pang et el. [30] and the spacewire architecture presented by Ermont et al. [10].

The implementation presented in this thesis is done in Ada 2005 with the Ravenscar restriction to make it easier to verify requirements in a model-checking tool. Related to this is earlier work done in UPPAAL regarding Ada software and tasking. Björnfot presented "Ada and timed automata" in 1996 [8] and Lundqvist et al. presented verification of an Ada RTS kernel with the Ravenscar restriction applied in four papers [22, 20, 19, 21].
2.3 Internet of Things

As part of this thesis work a first look at the field of Internet of Things is included and more specifically the different application layer protocols available. This section lists the main other protocols and frameworks available for use in the field of "Internet of Things" which could be related to the future of SPA and the "Virtual Network Protocol" fork.

Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP) is an application layer protocol going through standardization process in the IETF working group "core". The publication of the CoAP draft as a proposed standard is imminent as the document has reached "Authors’ Final Review" as of May 2014 [13].

CoAP is designed for highly constrained nodes, that is nodes with a 8-bit microcontroller and very little ROM and RAM. Added to the hardware constrains the microcontrollers are often connected to lossy networks with low bandwidth. The protocol is a request-response protocol that relatively easily can be proxied to HTTP as it works with URIs in the same manner [9].

MQ Telemetry Transport (MQTT)

Originally invented in 1999 MQTT is a publish and subscribe, lightweight protocol designed for highly constrained nodes, just like CoAP. It runs over TCP/IP so it expects a network stack on all devices. A version of MQTT called MQTT-SN for "Sensor Networks" is a branch of the protocol dedicated for networks without TCP/IP capabilities [28, 27]. The main protocol is undergoing standardization process in the OASIS organization, MQTT Working Group [29].

AllSeen Open Source project

From the beginning developed by Qualcomm as the AllJoyn project, the AllSeen Open Source Project is now run as a Linux Foundation Collaborative Project. It is a framework which offers service discovery, remote procedure calls, security and a lot of other features independent of the transport layer used. It is a framework with a core written in C++ with bindings to multiple other languages and runs on a wide variety of platforms from embedded to desktop devices [2, 3].
BACnet

In contrast to CoAP, MQTT and AllSeen Open Source Project the goal and primary concern for BACnet is building automation. BACnet, that is building automation and control networks, was originally started as standardization project within the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE). After publication of the american standard it has since been published as both an european standard and an ISO standard [6].
Chapter 3

Method

This chapter describes the method used throughout the thesis work. First the literature study and then the method used for designing, modeling and implementation.

3.1 Literature study

To get a general understanding of the field and current state of the art, a literature study was conducted. The result of the study is presented in chapter 2. The literature study for Space plug-and-play Architecture and the Model Checking was conducted by selecting different key terms related to the topics and searching for them in two different search engines. The first one was Google scholar and the second one was the "Discovery" search engine available to the students and employees of Mälardalen University. The Discovery search engine search through multiple databases such as IEEE Xplore, ACM Digital Library and SpringerLink. For searches in the Discovery search engine only articles with full text available was requested.

In Google scholar the first 20 results were looked into and the first 40 results given by Discovery search engine was looked into. Only results with full text PDFs were selected from Google Scholar.

A first read through of abstracts was conducted to filter out unrelated papers followed by a read through of introductions and conclusions for another filtering round. The key terms selected are shown below.
3.1.1 Space plug-and-play Architecture

The key terms chosen were "Space plug-and-play Avionics", "Space plug-and-play Architecture" and "why Space plug-and-play Avionics".

3.1.2 Model Checking and UPPAAL

The key terms chosen were "UPPAAL", "UPPAAL Modeling" and "UPPAAL Modeling Ravenscar".

3.1.3 Internet of Things

"Internet of Things" is a broad field for research and development. As this is only a first look at the topic no specific key terms were selected instead previously known protocols were selected for a short presentation in section 2.3.

3.2 An Agile Unified Process

For the development process inspiration was taken from the Agile Unified Process presented by Larman in "Applying UML and Patterns" [17]. As the SPA software is a framework no user stories nor use-cases was created instead the different SPA standards were used as a feature requirements list.

Throughout the spring regular meetings took place to discuss design issues with the author of a related thesis work, Nils Brynedal Ignell. Major parts of the design and implementation have been created in collaboration. The implementation in Ada with ravenscar restriction in combination with the UPPAAL model of the system have been created in small increments to reach their final state.
Space Plug-and-Play Architecture

Space plug-and-play Architecture (SPA) is a set of standards that define how different players can create plug-and-play components that can easily be assembled to form a complete satellite for a specific mission. The standards are published by American Institute of Aeronautics and Astronautics (AIAA). The standards range from how power supply should be handled to how components communicate in the application layer, focus in this report is put on the software parts.

To describe how a SPA network works some terminology needs to be defined. All nodes in a SPA network are called "components", there is no differentiation between software and hardware components. Each component has a "Component Universally Unique Identifier" (CUUID) which is 128 bit long. Each component also has an "Extensible Transducer Electronic Data Sheet" (xTEDS) file that describes respective components capabilities. To be able to support different underlying link layer technologies such as Ethernet, Spacewire and one-wire, most functionality has been put in the application layer. This means that a SPA network can be built with any combination of link layer "subnets", for example a SPA network can consist of two Spacewire subnets and one Ethernet subnet.

The important concepts to remember is that each SPA component is connected to one or more SPA subnets, on each SPA subnet there is a subnet manager (SM-x) that acts as a gateway to other SPA subnets and when SPA subnets are linked together they form a SPA network. To make things clear, a single SPA subnet with a CAS, LS and SM-x is also a SPA network.

Spacewire and Ethernet have been mentioned earlier, another SPA subnet
is the SPA Local Subnet. Each processing node that can have multiple SPA components running on it has a "SPA Local Subnet Manager" (SM-L). Components within a processing node should do inter-process communication over UDP/IP [36].

Components communicate with each other with the help of "logical addresses" that each component receives during boot up. It is the responsibility of the "Central Addressing Service" (CAS) to hand out logical addresses to all components in the SPA network through the connected Subnet Managers. After a component receives its logical address it can share its xTEDS file with other components. It is the responsibility of the "Lookup Service" to keep track of the capabilities of each component in the SPA network. For a detailed overview of the discovery process see section A.2.

To describe what a SPA network can look like figure 4.1 shows a minimal SPA network which consists of a CAS, LS, a sensor and a monitor application, all running on the same processing node and in the same local subnet managed by the SM-L.

![A Processing Node Diagram](image)

Figure 4.1: A processing node running a minimal SPA network.

A more complex SPA network is shown in figure 4.2. Three different processing nodes are connected to each other over a SPA-Ethernet subnet. The CAS, LS and Monitor Application are now running on different nodes and the sensor application is located on its own embedded device.

For a detailed view on how SPA works the SPA standards is the best source for information. The SPA standards and drafts that specify software requirements are the Logical Interface standard [32], Networking standard [33], Local Subnet Draft [36], Ontology Standard [34] and System Capabilities Standard [37].
Figure 4.2: A SPA network with multiple subnets.
Chapter 5

Designing and modeling the system

This chapter includes the UPPAAL model in conjunction with corresponding UML diagrams describing the implementation done in Ada. The UPPAAL model is used for the validation and verification of the system that it meets the requirements set on the system. The connection from the UPPAAL model to the UML diagrams presented here are to show that the implementation corresponds as close as possible to the models verified.

The creation of the UPPAAL model presented in this chapter focus on the component and service discovery process. After the initial discovery and configuration process has been completed, the system continues in normal operation mode with an extra set of SPA message types that has not been modeled nor implemented. Detailed sequence diagrams over the component and service discovery process is shown in section A.2. The process from start to the final result is presented over a few iterations.

UPPAAL and SPA terminology

To better understand the following chapter a few words on the terminology used is necessary. A SPA component is either a SPA hardware device or a SPA software application. In the design of the system a SPA application is implemented as an Ada task. In UPPAAL the same application is modeled as an automaton. The different SPA applications such as CAS, LS and SM-L are modeled as different automata, together they form a network of automata which forms the UPPAAL model. A GUI is used to create the different automata but for more advanced features such as a queue or an object the underlying C-like language can be used instead.
The reason for this is that an automaton of a queue can easily become too complex to work with or verify.

The verification of the UPPAAL model is done by creating verification queries and sending them to the verification server that runs in the background, all is done through the UPPAAL GUI.

5.1 First iteration

The first UPPAAL model was created after designing the basic structure of the system. Each application will do the same thing which is receive, handle and send messages. The difference between the applications is which messages each application understands and which messages the application will ignore. A processing node will at most have one CAS, one SM-L and one LS but can have any number of applications running next to them. This inspired the first iteration of the UPPAAL model. The application automaton should be possible to instantiate multiple times in UPPAAL but the other more specific automata can be used only ones. The basic SPA component overview is shown in figure 5.1 which is what the UPPAAL model is to resemble as close as possible.

![A Processing Node](image_url)

**Figure 5.1:** A Processing Node overview with core SPA components.

A basic application

The basic functionality of an application is that it should be able to identify itself on the network, share its capabilities and request information from
other applications. An application is only connected to one subnet at a time. The limited functionality of an application makes for a basic two-layered design where the upper-layer alters its behaviour depending on incoming message types and the lower layer is only responsible for sending and receiving messages on its subnet. The application sends out Local_Hello message during boot-up and receives a Local_Ack response, all messages are modeled as transition between different states. It then receives a logical address with the Assign_Address message. As final step it receives a Probe_Request from the Lookup Service to share its capabilities. Figure 5.2 shows the UPPAAL model.

![Diagram](image1.png)

**Figure 5.2**: UPPAAL model of a first iteration application. Light blue text indicates synchronization channels, dark blue text indicates updates to variables and green text indicates guards in UPPAAL.

### Central Addressing Service (CAS)

Specifics for the CAS is that it assigns an address to itself so there is no need for processing of Assign_Address messages. As addition to basic application functionality the CAS sends out address blocks to the requesting subnet managers. Figure 5.3 shows the UPPAAL model.

![Diagram](image2.png)

**Figure 5.3**: UPPAAL model of a first iteration CAS.
Lookup Service (LS)

The additions the LS brings, is the processing of Request_LS_Probe, Probe_Reply and Probe_Request messages, for service discovery purposes. Figure 5.4 shows the UPPAAL model.

Local Subnet Manager (SM-L)

In contrast to the other components the Subnet Manager manages most of the interaction done in the subnet. It is responsible for answering all Local_Hello messages during component discovery and replying with an acknowledgement as well as with a logical address assignment. As a last step it tells the LS to send out Probe_Requests to all active components on the subnet. Figure 5.5 shows the UPPAAL model.
Conclusion from the first iteration

The CAS, LS and SM-L have specific functionalities so they can only be used once when setting up a complete network of automata. The application model can be instantiated multiple times as was the goal. At this level of abstraction everything looks fine, though the main goal with the UPPAAL model is to be able to verify that the system behaves as expected and can not reach erroneous states or deadlock. As the first iteration of models only shows the boot-up process it will eventually reach deadlock when everything is set up. This is a drawback as it does not relate very well with the actual design of the system after boot-up. This calls for another iteration and improvement of the different UPPAAL automata.

Before heading to the second iteration it is time to take a closer look at the UPPAAL automata and see what other parts need improvements, as it should resemble the actual system as much as possible.

First of is the current automata use transitions to model sent and received messages directly between the different automata. When a transition is taken to send a message the receiving automaton takes a corresponding transition to receive the message at the same time. This is in contrast with the implementation where all messages go through a queue. The second iteration of UPPAAL model should be modeled with proper queues that can be scaled in size.

The model only work with a set of eight different message types and in the current design of the UPPAAL automata it is hard to add more types of messages. The second iteration should make it possible to add and remove specific message types with ease.

5.2 Second iteration

To improve the UPPAAL model a more detailed design is required for the system. From the first iteration it is clear that the basic application can be designed in a generic fashion. The CAS and LS are more specific instantiations of the basic application and the SM-L has substantial differences. For easier maintenance of the models during the second iteration it would be valuable if the UPPAAL model could be created in a modular way. Two main automata would be the ”basic application” and the ”SM-L”.

Figure 5.6 shows the UML component overview after the second iteration. The design of the interfaces between the components makes it easy to interchange which message types each ”running task” (application) can handle.
and how it communicates. A simple application that is only connected to one subnet does not need the routing component. The routing component is specifically created for the SM-L to be able to route traffic within its own subnet and to other subnets.

The design of the communication package makes use of the "composite pattern" where routing objects and com objects all inherit from the same interface and where the routing objects are of composite types. Figure 5.7 shows the difference between a simple application and the SM-L. The use of the composite pattern makes it possible for an application to easily swap out how it connects to one or multiple networks. In the implementation presented in this thesis this is done at compile time.

The queues

In the implementation of a local SPA subnet presented in this thesis, communication is done through protected objects in Ada. Each protected object includes two queues, one from the application to the SM-L and one back. To model this in UPPAAL the C-like language could be used to create the actual queues in the background of the queue automaton. The finished queue is shown in figure 5.8. Each queue is instantiated for each SPA application including the CAS and the LS.
Figure 5.7: The difference between a simple application and the SM-L.

**A basic application**

After modeling the queues the application was up next. The application automaton is shown in figure 5.9. In addition to the queues implemented in UPPAAL C-like language a "Message" struct was also defined to model a SPA message, including its header. Each message is modelled with a source and a destination address as well as an opcode identifying what type of message it is.

A key assumption during iteration two was that upon receiving a message each application should send a response immediately. This was modeled in all automata including the one for the application. Upon receiving a message with the opcode Assign_Address or Probe_Request another message was sent in response before taking any other action. This introduced deadlock scenarios if the queue back to the SM-L was full the automaton for the application could not take any transition and if the queue to the application
was full as well, both the SM-L and the application would be waiting for each other to empty their respective queue.

A solution to this that was tested allowed each application to throw away all incoming messages if no response could be sent back out. In turn each application had to keep track of which messages it had sent out and which responses it expected. After a few improvements this solution turned out to be too complex, resulting in a model that was hard to understand and time consuming to verify.
Second iteration conclusions

As too many problems arose with the second iteration only the application and queue automata are shown in the report. Even though the end result of iteration two was not good enough improvements were made throughout the iteration. The most significant improvements are presented in the list following this paragraph.

- Iteration two added the queue automaton which made it possible to model if and how the size of the queues would effect the system.
- The addition of the Message struct made it possible to model different types of messages being sent in the system.
- The logic throughout the system was vastly improved.
- The possibility to vary the number of applications was kept.

5.3 Third iteration

The main focus of the third iteration was to remove the possibility of deadlocks. The underlying cause of deadlock was the tight coupling between receiving a message and sending a response. To loosen up the coupling different approaches could be taken. One solution was to add time constraints such that if the outgoing queue was full the automata would wait for a set amount of time and then retry. It was early on apparent that this would not improve the deadlock situation.

A second solution and the one chosen was to make the assumption that when receiving a message the information is stored locally and then the application return to its idle state. From the idle state the application can then make a choice whether to receive another message or send a new message. The tight coupling was by this change completely removed.

The automata from the third and final iteration are shown in figures 5.10, 5.11, 5.12, 5.13, 5.14 and 5.15. Together with the queue automaton (fig. 5.8) from iteration two they form the final model of a SPA Local Subnet that use Ada Protected Objects as communication channel.
5.4 Verification

To verify that the system is free of deadlocks and livelocks the UPPAAL verification queries presented in the end of this section were used. As the automata only test the component and service discovery process the goal is to reach a deadlock state in the end. To verify that it is only the final deadlock that can be reached the queries verify that all automata are in idle states and the appropriate variables have been set (all applications have been properly registered in the network).

The livelock query makes sure that all paths eventually reach the final deadlock state. Both verification queries are shown below.
Verification query for deadlocks

\[ A[	ext{deadlock}] \implies \]
\[
\begin{align*}
\text{CAS. Idle && SM\_L. Idle && LS. Idle &&} \\
\text{LS. Received\_Probe\_Reply [CAS\_Address] == true &&} \\
\text{forall (Application\_ID: Application\_ID\_Type)} \\
\text{Application (Application\_ID).} \\
\text{Local\_Ack\_Received == true &&} \\
\text{Application (Application\_ID).} \\
\text{Assign\_Address\_Received == true &&} \\
\text{Application (Application\_ID). Idle &&} \\
\text{LS. Received\_Probe\_Reply [Application\_ID] == true}
\end{align*}
\]

Verification query for livelocks

\[ A\lnot\text{ CAS. Idle && SM\_L. Idle && LS. Idle &&} \\
\text{LS. Received\_Probe\_Reply [CAS\_Address] == true &&} \\
\text{forall (Application\_ID: Application\_ID\_Type)} \\
\text{Application (Application\_ID).} \\
\text{Local\_Ack\_Received == true &&} \\
\text{Application (Application\_ID).} \\
\text{Assign\_Address\_Received == true &&} \\
\text{Application (Application\_ID). Idle &&} \\
\text{LS. Received\_Probe\_Reply [Application\_ID] == true}
\]
Figure 5.12: Iteration 3 - The send loop part of the SM-L automaton

Figure 5.13: Iteration 3 - The routing loop part of the SM-L automaton. All messages with a destination that is not the SM-L are routed through this loop after first being received in the receive loop.
Figure 5.14: Iteration 3 - The Central Addressing Service automaton
Figure 5.15: Iteration 3 - The Lookup Service automaton
Chapter 6

Result

6.1 Model verification

The model presented in chapter 5 was verified with the verification queries presented in the same chapter.

- For deadlock verification the property was satisfied.
- For livelock verification the property was not satisfied.

6.2 Implementation

The implemented SPA framework code called ”Virtual Network Library” is available on Github at https://github.com/virtualnetwork/vn-lib (including the third iteration and final UPPAAL model). The modular design is shown in figure A.1 and the source code compiles with the Ravenscar restrictions. Efforts have also been made to limit the amount of dynamic memory usage.

Example implementation is available in the ”examples” directory in the git repository. Below is a series of images that show example output from a test run. The example output is the same process as is described in section A.2 with UML sequence diagrams.
Figure 6.1: First the configured components start and send out Local_Hello messages which are processed by the SM-L. SM-L replies with Local_Ack to all other components and a specific Request_Address_Block to the CAS.

Figure 6.2: The CAS replies with Assign_Address_Block which is processed by the receiving SM-L. After this the SM-L starts to hand out logical addresses to all components in the local interconnect. The logical addresses are received by all applications. As a last step the SM-L sends out Request_LS_Probe messages to the LS for all components on the local interconnect. This last step is the end of the network discovery process and the start of the component data capabilities discovery process.
Figure 6.3: The LS receives the Request_LS_Probe messages and sends out Probe_Request messages in response. The Application and CAS receives their respective Probe_Requests and replies with a Probe_Reply. The example output finishes with the LS receiving both Probe_Reply messages.
Chapter 7

Conclusion

Presented in this thesis report is a model-checked implementation of a new subnet local adaptation of the Space plug-and-play Architecture. It has been implemented as the Virtual Network Library [40] for the fork of SPA called "Virtual Network Protocol". Instead of UDP/IP for IPC, Ada Protected Objects are used to share data between different Ada tasks.

When model-checking a system to verify that it behaves as expected it is important to get the model to resemble the implementation as close as possible and vice versa. It is a network of two development artifacts that prove that the system behaves according to the specifications. In this thesis the specifications are the SPA standards and the implementation is the "Virtual Network Library" implementation.

In chapter 5 the process that took place to reach the end result regarding the UPPAAL model and the implementation is shown. Going further with an even more detailed model would go into unnecessary detail such as how the routing tables or the CAS table are implemented. As the different tables are not going to hold any substantial amount of data, the time to do a lookup in the different tables are negligible. This among other things led to the conclusion that the level of abstraction of the model is at the right level for the purpose of this thesis.

The verification of the model shows that the only way to reach a deadlock is in the end state as expected, the problem comes with livelock. The verification query for livelock shows several different scenarios on how the system can reach a state of livelock. A very simple example is in the beginning when an application sends a Local_Hello to the SM-L and the SM-L receives that message. When the SM-L is back in idle state the system reaches a livelock scenario because the application will keep sending Local_Hello messages until it receives a Local_Ack response. Similar scenarios arose throughout the
development of the model, for example when the SM-L sends a request to the CAS for an Address Block. Unless the CAS responds with an Address Block the SM-L would keep sending Address Block Requests forever. The solution to this scenario was to limit the SM-L to only send one message and make the assumption that all messages sent will be received. As communication is through Ada Protected Objects this is an acceptable assumption to make in the current implementation.

The same solution could be used to the livelock problem with Local_Hello messages. The drawback would be that if the application would be moved to any other type of subnet, the assumption about guaranteed delivery would not hold. In the end some kind of time based retry algorithm could be used, such as to resend the Local_Hello message until a Local_Ack is received without resending too often. To be able to model the different SPA applications in more detail they could be separated out and connected to an automaton representing the environment. This would allow testing of specific timings in different scenarios.

**Future research - maintainability**

One major consequence of moving away from UDP/IP as IPC is that the applications running on the different processing nodes no longer are modular in the sense that it is easy to add and remove applications on a processing node in a running system. When using Ada Protected Objects all SPA applications must be compiled together into one binary and started at the same time. This leads to future updates must replace the entire binary and reboot the system. The trade-off between Ada Protected Objects and a run-time modularity is something that need to be prioritized per project. For robots or satellites that are not connected to internet the increased reliability of protected objects may be a good trade-off. On the other side devices connected to internet that always must be up to date security wise to protect against intrusions, the modularity may be prioritized. An example to this could be a refrigerator that is connected to internet and a vulnerability is found in its transport layer security code. Instead of updating the entire running binary it would be enough just to update the specific module that exposes the vulnerability. Future work should take a deeper look at this trade-off, when is it worth the effort to use Ada protected objects over the modularity of UDP/IP for IPC?
Future research - security in general and confidentiality in particular

For closed-circuit networks like remotely operated vehicles and some robots, security may not be that important but for the vast majority of devices security needs to be introduced. As soon as a device is connected to the Internet and starts to communicate with others it will need some kind of authentication mechanism to allow access to its own data, it will need authorization mechanism to be able to differentiate between other devices regarding which device should get access to what data. Added to this is transport encryption (confidentiality) to prevent pervasive monitoring and the possibility of disclosing private information to active listeners (man-in-the-middle attacks). How to add security to a SPA network with open access to internet is an important step forward to make SPA a viable option in the world of Internet of Things.

Future development - dependability of the development process

When starting this thesis Ada was chosen as programming language because of its track record with software that must be dependable to high-levels. Different ISO standards and white papers from AdaCore introduce good programming practices, how to develop good quality software and how to use object-oriented programming in the field of safety-critical applications. The approach all these papers use is that the developer and/or architect need to decide upon a few things first, before starting development.

First of the project must know what kind of safety level is expected from the system. This relates directly to how much testing, validation and verification must be done throughout the development cycles and in the end when delivering the system. Testing, validation and verification techniques then influences the choice of programming features that can and more interestingly, should be used.

During the development and design phases in this project quite a lot of discussions went into which level of safety that was expected. With a specific product in mind and its requirements this would have been less time consuming. In the end, without specific safety requirements focus was put on which Ada features that were thought to be useful and which ones should be avoided. The reasoning behind this is that the fewer Ada constructs that were used the easier it would be to validate and verify in the future (and porting the system to new architectures). Future work should take a deeper look at the different safety standards that may be of interest and come up with firm programming guidelines that can be used during development.
Bibliography


Appendix A

System Design

This appendix shows the overall design of the system in UML layer, class and sequence diagrams.

A.1 Virtual Network Library implementation, high-level design

To get one simple application up and running in a ”Virtual Network” the application’s settings need to be defined in the Application.Information component. This ranges from everything from CUUID to other custom settings that may be needed. The Application.Information component also keeps track of the application’s logical address during run-time and supplies different helper functions, one of which is to fill in VN.Message header information.

The next step is to add connectivity. If the application only runs on a processing node or on a single subnet all that is needed is a VN.Communication.Com implementation of that specific subnet type. The more advanced use case is a subnet manager that manages its own subnet and is connected to a Local Subnet Manager on the processing node it runs on. As this requires connectivity on more than one subnet a Protocol.Routing object will be needed to route traffic between subnets. The Protocol.Routing object is a composite type of the VN.Communication.Com interface and can therefore add multiple subnets during startup procedures.

All communication on a processing node through Ada Protected Objects goes through the Subnet Manager as a router. This leads to all applications on a processing node only has to instantiate one connection. This is done
by first instantiating one Protected Object (PO) through the
\textit{VN.Communication.PO} component. An access to this PO can then be used
as input parameter when creating the two \textit{VN.Communication.PO\_Wrapper’s},
one for the application and one for the local subnet manager.

The application itself is written as an Ada task and due to Ravenscar limita-
tion must be built from the ground up for each new task. Best way to start
is to copy an example task from the ”examples” directory in the repository
[40].

\textbf{A.2 Component and Service discovery process}

The following sequence diagrams show the basic component and service
discovery process for a local subnet using UDP/IP for inter-process commu-
nication described in SPA Local Subnet Adaptation Draft [36] (the sequence
of messages sent back and forth is the same for a Local Subnet with Ada
protected objects).
Figure A.2 starts with the high-level view of a complete SPA network. Figure A.3 continues with the specifics of a SPA Local Subnet. Figure A.4 shows the specifics on how a SPA Subnet Manager acquires an address block from the CAS and after that the SPA Subnet Manager assigns a logical address to each component it is aware of in figure A.5. The last step is the component capabilities discovery process where the LS collects information about all known components on the network, this is shown in figure A.6.

Figure A.2: Overview of the discovery process for a SPA network.
Figure A.3: The basic discovery process for a SPA Local Subnet.
Figure A.4: How the local subnet manager acquires address blocks from the CAS for its own subnet and for other connected subnet managers.

Figure A.5: After the SM-L has received Address Block it starts to assign logical addresses to local components it is aware of.
Figure A.6: The last step in the discovery process is to discover which components have which services.