Open University

TM470 – IT & Computing Project

Python UDS Manager

Final Report

Richard Clubb

Supervised by:

Dr. Samir Al-Khayatt

Abstract

This is the final report

Table of Contents

1. Introduction

All road vehicles produced today use some form of diagnostics for inter-module or off-board diagnostic communications and there are a variety of standards which cover specific areas of the industry and the Unified Diagnostic Standard (UDS) is used for most personal vehicles.

Embed is an automotive systems engineering consultancy which provides solutions for the entire stack of engineering problems, from systems engineering to hardware design. They currently have a tool nicknamed 'Gordon' which is used for UDS communication, specifically programming Electronic Control Units (ECU/ECUs), it was developed as part of an old project and was implemented to interface with a specific ECU with a single configuration. It was later used to provide solutions to a customer who did not have access to the more expensive industry standard tools for UDS diagnostics, but at this point it needed to be re-compiled with a different configuration which created a fork in the source tree.

The engineering director at Embed wants to create a more flexible tool which could provide basic programming and diagnostic facilities to our customers for free so that those who only required a small sub-set of the industry standard functionality did not have to buy expensive tools.

This project is a software engineering exercise to develop Embed's new tool. It will use their systems engineering philosophy and practices which are developed using the Systems Modelling Language (SysML) and Model-Based Systems Engineering (MBSE) processes. Embed are developing their own systems engineering framework called the Embed Architecture Framework (EAF) which will be used to perform the higher-level requirements and architecture. The EAF was designed to provide a framework for Systems Engineering and this project will exercise a more software focused usage of the framework.

1.1. Terminology and Formatting

Throughout this report the use of *italics* is intended to highlight names of methods or variables relevant to the code.

Code snippets will be written in `Courier New` and intended, this may cause the code to wrap across lines.

A Glossary of Terms is included in Section 12.

2. Problem definition

During an ECUs lifecycle the need for diagnostics exists, this can be in the early stages of implementation where an engineer may need to query signals and values, or at the deployment and maintenance stage where the ECU may need to be re-programmed with a newer version of its software.

UDS was developed to create a standardised protocol for communication and covers a wide variety of use-cases such as; parameter diagnostics, memory management, configuration programming, application programming, input/output control and security access. There are a variety of industry tools which support UDS such as Vector Informatik's CANape and Bosch's ETAS, as well as some open source tools such as BUSMASTER (also written and maintained by Bosch). However, the industry tools are very expensive both for the hardware and software, and the licenses are not only expensive, but very complex, and BUSMASTER is very difficult to use and modify. Neither provide an ideal solution for smaller businesses or people who do not want to invest in tools which may not be able to pay back their investment.

Embed have an existing tool for UDS communication (called Gordon) written in C# which used UDS over UDP as the transport protocol[1]. This was written for a previous project where the intention was only ever to interface with a single ECU (An E600) under a very well-known set of use-cases. This tool was later expanded so that it could use a Common Area Network (CAN) interface to communicate with both the E600 and E400 ECUs to provide a free alternative to reprogram the ECUs without the need for expensive industry tools. These ECUs have a very similar bootloader[2] and configuration and do not require different diagnostic definitions.

Unfortunately, it was later re-purposed again to support a completely different ECU, this required small changes to be made to the code, but meant the entire application had to be re-compiled and created a source tree divergence. It became quickly obvious that this tool was quite limited and required significant re-work to work with the changing use cases. The scope of the changes was very small, but as the software was not architected for this purpose the system could not extend (Martin, 2018).

A basic analysis was performed and it was determined that re-factoring the C# code would be more complicated and less flexible than re-implementing the solution. The original tool used an extension called Iron Python which provides the ability to run an external Python script, and to be able to pass in libraries and variables for the script to use. This functionality allowed users to create scripts to run which did not require re-compiling the program each time a change was made, this provides a level of flexibility in its use. One script could be used to verify part numbers, while another could re-program the ECU. However, Iron Python only supports Python 2.7 which does not receive any further support from Python, and many libraries no longer support it.

The idea was raised that a new tool could be developed to act as a general purpose UDS communication tool, and would allow users to interface with ECUs easily.

[1] It was actually more complex, the E600 is a telematics unit which uses GPRS communication. The ECU set up a connection channel to a server and the server communicated with the ECU via a Can Transport on UDP over the GPRS network to be able to interface with the unit. It isn't actually a Diagnostics over IP (DoIP)

[2] A bootloader is a common part of most automotive ECUs, it provides a fixed application which resides in the ECU and can not be modified. It provides a methods to re-program the ECU easily and a failsafe environment if the application fails to program. Generally it will have two parts, a Primary Bootloader (PBL) and a Secondary Bootloader (SBL). The PBL is ROM resident and is always in the ECU, the SBL is RAM resident and is used to re-program the ECU as many microcontrollers can not be re-programmed while running in ROM.

3. Initial Investigation

The problem domain for UDS communication is well known and the use-cases are quite well defined. The UDS protocol is defined in the ISO 14229 standard (ISO, 2013) and covers a wide range of defined functionality, as well as leaving parts of the standard open for manufacturers and suppliers to implement their own specific solutions. It is extended by the ISO 15765 standard which provides the information for how to use UDS over other communication protocols such as CAN (ISO, 2016), LIN or FlexRay where the transmission payloads are smaller than the UDS payload

At the beginning of the project the idea that this would be released as an open-source tool was suggested, and this means that the general scope of usage may be larger than Embed's initial ideas. From this an analysis of a popular maker site (Hackaday, 2018), (Anool Mahidharia, 2017) was performed to see how people would likely use this software. This helped to build up an idea of the contexts and use-cases which would aid in the requirements analysis.

3.1. High-Level Interface Concepts

At a high level, the user wants to be able to easily interface with the ECU. Dealing with messaging at a CAN Protocol Data Unit (PDU) level is very unfriendly as the signal definitions follow no standard and can change per vehicle line, model and model year (engine speed could be engineSpeed, engSpeed, engineSpd, n_rpm which have all been seen in configuration files).

UDS abstracts its functionality to a set of defined services and sub-functions, while the CAN Transport Protocol (CAN TP) handles sending large data payloads. These follow a hierarchy where the ECU implements services and those services have specific sub-functions, some of which are pre-defined by the standard and others are proprietary. It was thought that the user would wish to use this hierarchy to work with the ECU, so a high-level interface concept was investigated.

The diagnostic database files are referenced relating to the service and the sub-functions registered for those services. Following this convention, a generic template was proposed.

```
[device].[service](sub-function, parameters)
```

E.g.

```
a = E400.ReadDataByIdentifier("ECU Serial Number")
```

Where DiagIdentifier.ecuSerialNumber is an enumeration which would translate to the appropriate Diagnostic Identifier (DID)

would retrieve the ECU Serial Number from the E400 or

```
E400.WriteDataByIdentifier("ECU Serial Number", "SN00000000001")
```

Would write the new serial number to the E400.

This creates a well-structured and abstracted interface for the user to control. It allows for multiple device classes to be instantiated as well as specific configurations for those classes. For example, if "ECU Serial Number" has a different definition for the E400 and E600 (e.g. the length is different) then the E400 will encode it with its method and the E600 with a different method.

This creates a problem which is that the methods and classes are not statically defined and depend on the configuration passed to them. This is very common with compiled code for embedded systems, CAN Signal Database Definitions (DBC) and LIN Definition Files (LDF) are usually the single source for the signal configuration, and are parsed to create configuration files, and these are used at compile time. The application knows about the signal, but the position of the signal in the message and the behaviour of the message on the bus is abstracted away.

Dynamic function definition and creation is not advised as it proves difficult to test, and will have to be considered during the design phase.

3.2. Existing Solutions

At the beginning of the project an investigation was performed to see if there were any other projects that could be improved rather than starting from scratch. There were a few small projects but the most developed was the udsoncan project (Lessard, 2018). It is written in Python and supports all of the ISO 14229 services.

Unfortunately, it has a few fundamental drawbacks. Its data model is based on the service specification, rather than any implementation of the service. While this design decision does have some advantages, it poses some difficulties as well. Some of the encoding is performed by the service class, such as the creation of a DID list for the Read Data By Identifier service (ISO, 2013, page 101) but it has no ability to decode the actual data based on the diagnostic definition. It mainly provides a mechanism to encapsulate the supported positive and negative responses for the service, but if a user is having to manually code the DID into the service and decode the response then it doesn't save effort. It is also built upon the socketcan (Linux Kernel, n.d) interface which is native to Linux but not windows and leverages the CAN TP implementation in the Linux Kernel code. This would create a difficulty when implementing the other required hardware interfaces, and provides no support for Windows. Due to these constraints it was decided that this would not be a good starting point for the system.

3.3. Legal & Ethical Considerations

The software poses no ethical or moral issues that can be identified. It uses industry standards which are publicly available, and is commonly used in the automotive sector.

The sponsor Embed has been consulted and has given permission for the software to be released for free as open source. The software was never intended to be used for monetary purposes.

It is uncertain at this time if the software would incur any liability, and this is being investigated further. Generally, the UDS Server (ECU or device) has liability concerns as it may be implemented on a safety or mission critical piece of hardware, but the client software is simply an interface to the server and thus should not be covered under the same liability concerns.

3.3.1. Open Source Licencing

This software is intended as a free tool for engineers to use to avoid expensive tooling. The initial discussions led to the idea that it should be an open source tool enabling clients to make changes easily. Releasing the software as Open Source seemed logical and though needed to be given to the licence the software is to be released under.

The popular choices are the MIT Licence, the Apache 2.0 licence and the GNU General Public Licence (GNU GPL). The MIT and Apache licences are generally permissive, while the GNU GPL is more restrictive, placing heavy constraints on the modification. This has been known to put developers off modifying the software as it demands that any changes must be made public. The advantage of this is that changes should make their way into the main development stream, thus enhancing the overall software quality.

The MIT licence was chosen for this project as it is popular and permissive. It does not stop later versions of the software being released under other licences. The complexity of choosing an open source licence is not fully understood, and some advice was sought from developers at Embed on which licence to choose, the general consensus was the MIT licence, but this is an avenue of enquiry that may need to be revisited at a later stage in the project, probably before the V.1 release.

3.4. Lifecycle

Initially a traditional Waterfall lifecycle (Royce, 1970) was chosen as the suitable model for the project. This was chosen due to the assumed simplicity of the project, the well-defined scope of the system, and the number of developers involved. The agile development methods favour interfacing with teams of developers rather than either a single team or individual. They are also favoured when the requirements are not as well developed or the system domain is not well understood as the process can iterate to a solution faster than a more rigorous Waterfall.

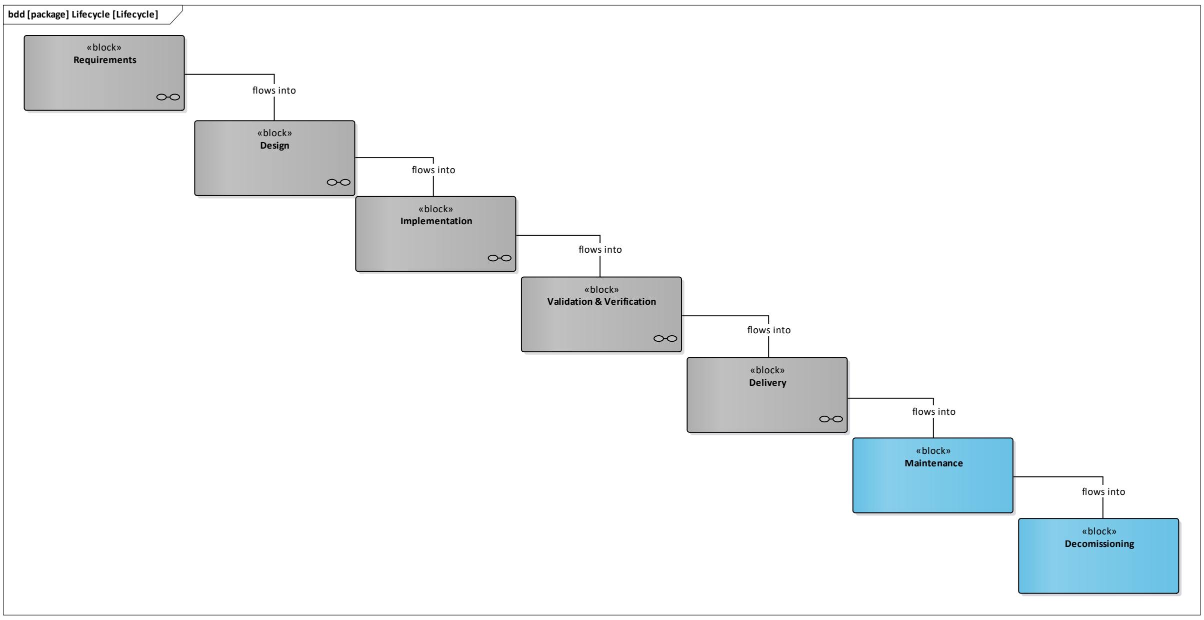


Figure 1 - Waterfall lifecycle model

The requirements phase would follow the Approach to Context-based Requirements Engineering (ACRE) (Holt/Perry, 2013) This process is followed internally by Embed for their projects. The design phase carries on from the ACRE process and the architecture developed creates the technological contexts for the lower levels of development.

As this project is being delivered by one individual, it was deemed unnecessary to go further than the initial systems requirements defining the requirements for the solution domain (Hull / Jackson / Dick, 2005). If the project had involved a larger team, then an Agile development methodology would have been preferred, and it would have been necessary to perform further component requirement definitions.

Test-Driven-Development (TDD) will be used in the implementation phase to make sure that the code is suitably covered by unit-tests and is tested against the requirements. This is not a typically Waterfall approach as TDD is its own development lifecycle, but the method for developing code after the unit-tests have been written is a sensible development decision (Osherove, 2005).

3.5. Project Maintenance & Planning

As this is intended to be released open-source a suitable release platform was required. GitHub is a popular repository and provides not only public platform for distribution but also for general source control, version control and project management. Embed generally uses SVN for their projects but they do not have the facility to open their servers to public contribution so a management concession was made for this project.

The use of GitHub also opens up several other options for documentation. The project uses a documentation page on readthedocs.io which integrates with GitHub. Using these two solutions also helps with the delivery and maintenance phases of the project as GitHub has a well featured project management page for issue and task tracking.

3.6. Security Concerns

As part of the system involves dynamically generated code, there is the possibility of securing holes being opened that could be exploited to gain access to the host system similarly to malformed SQL queries in web services.

The potential vulnerability is that an attacker could release a crafted ODX file which would generate malicious code in the host PC. It is unlikely that this is a potential attack vector as to make effective use of this the attacker would most likely need access to the source either on the hosted repository, or on the target machine and if this was accessible they would likely use that attack surface would be easier to manipulate, rather than use another level of indirection. This software should not need to be run at elevated permissions, so would be protected by the host OS security policy.

While the attack vector is unlikely, it is worth protecting against, so further investigation will have to be performed to check the XML file for malformed inputs, and the XML parsing code should perform validation on any of the text and integer inputs to make sure that they are valid. This is not going to be performed for the initial delivery as it is outside the scope of the report.

3.7. Language Selection

Gordon was written in C# and used Iron Python as an extension to provide access to run Python scripts, but as noted in the Initial Investigation section it was not suitable as it would require re-compiling the program for small changes which creates source tree forks as well as Iron Python using Python 2.7 which is officially out of support.

One need of the project is the ability to change the UDS configuration easily, and another is the need to produce scripts similar to Gordon to allow programming and communications flow. Python is a high-level interpreted language which would fulfil the second objective, and has the ability to parse UDS configuration files easily. (there are several XML parsing tools available). It is also very popular and has a large number of available libraries. As the language is high-level and very popular, it would allow users to modify the code to add functionality.

The primary inputs to this software are user written scripts, and diagnostic database definitions. The database definitions are either ODX or CDD, and both are written in XML with different schema. Python has a range of XML parsing tools which are easy to use and well documented. Python seemed like a good choice for the development of this tool.

4. Requirements

This section goes into detail on the requirements analysis and elicitation process for the project.

4.1. Requirements Process Overview

The ACRE process was used with a hierarchical model of requirements (Hull / Jackson / Dick, 2005) (Broy / Gleirscher / Kluge, 2009) and the process defines high-level requirements concepts as Goals, these are usually light in detail and large in scope. The next stage is the definition of the "contexts" to identify any interested stakeholders and then the goals are analysed from each of the contexts to define the Capabilities. When an initial system architecture is defined, the Capabilities are analysed from each of the system contexts and this then provides the system requirements.

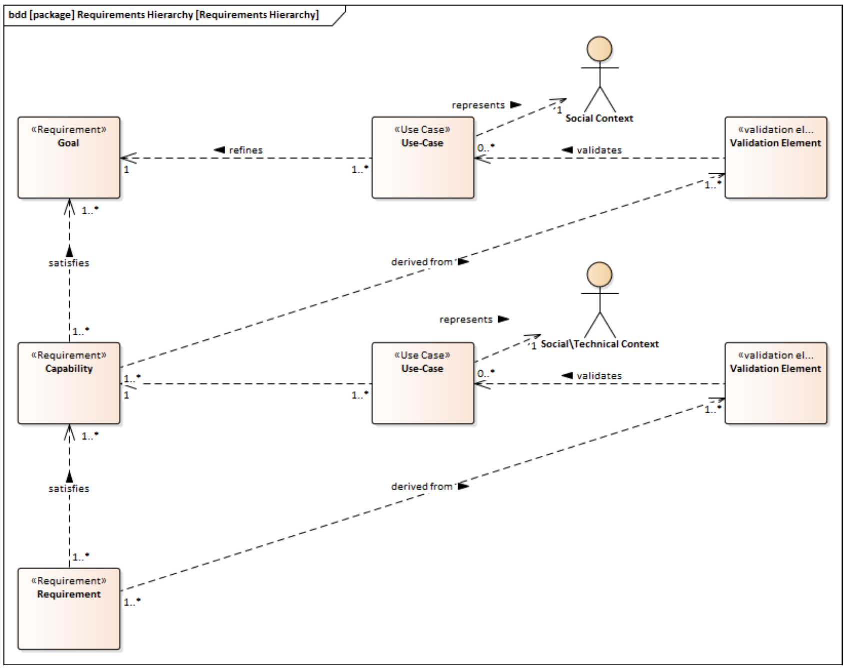


Figure 2 - Requirements derivation hierarchy

Each Requirement is refined by 1 or more Use Cases which represent different Contexts, at the top level these contexts are usually social, and the further down the hierarchy the more technical and concrete they become. Each of these Use-Cases is validated by a Validation Element which explains in more detail the exact use, typically these may be sequence diagrams, or parametric diagrams, external models or prototype code, or simply a detailed explanation. The next level of requirements is derived from these Validation Elements, and then directly Satisfies the Requirement at the level above. There must be a logical chain between the higher-level requirement and the derived requirement through a use-case and validation element for the satisfaction to be consistent.

For this project a Goal at the high level is analysed from a social context to produce a Capability which defines the high-level functionality for the system. These capabilities are analysed from the system context to provide the system requirements. The Goals, Capabilities and System Requirements for this project are listed in Appendix D.

Initially when this problem was discussed the Goal was for a replacement of the older utility with some modifications, but after some consideration it was decided that a full project would possibly solve problems encountered from other departments. A requirements elicitation was performed from different project managers in Embed to elicit the Goals of the project.

The main customer contexts were decided to be Vehicle Engineer, and Hobbyist. The Vehicle Engineer would be performing multiple tasks, such as module validation & development, programming, diagnostics, and vehicle maintenance. The Hobbyist would want to perform diagnostics and vehicle maintenance, but would likely not have access to the diagnostic databases. Some of the use-cases for these contexts overlap, but the general usage comes from a different perspective.

Along with the social contexts an analysis of the technical domain was also performed, to gain a greater understanding on how this software will work in the system domain.

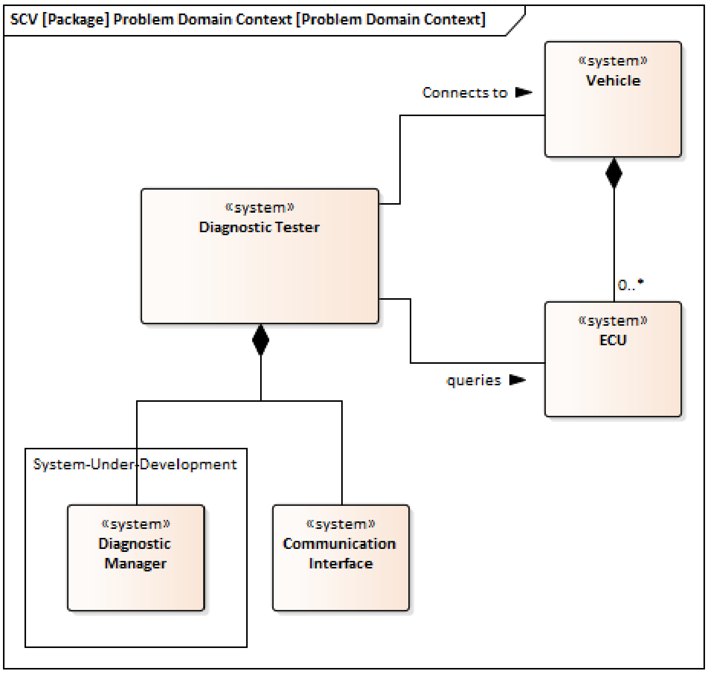


Figure 3 - System domain and system context

The System context was determined to be the Diagnostic Manager which is a software system of the Diagnostic Tester. It interfaces with the Communication Interface to provide communications with the Vehicle and ECU.

From this context the Capabilities were analysed to provide the system requirements which are detailed in Appendix D.5.

5. Design

Designing the architecture is an iterative process where the requirements are analysed and components are selected, new components are designed and investigated and then this learning is fed back and assessed to see if the solution is suitable. (Nuseibeh, 2001)

For the requirements to be implemented an initial architecture had to be created, each component was analysed and their function determined to see how they would satisfy the requirements. In more rigorous or safety-critical processes following ISO 26262 or ASPICE conformance there must be traceability to specific lines of code and functionality. The ACRE process can assist in by defining technological contexts which are used to analyse the system or component requirements. Further requirements can be derived and then code linked directly to those more specific requirements.

This process is very rigorous but is far too cumbersome for this project as there is no need for that level of requirements traceability.

5.1. Initial Architecture

The initial architecture is shown in Figure 4. This is based on an initial analysis of the system requirements.

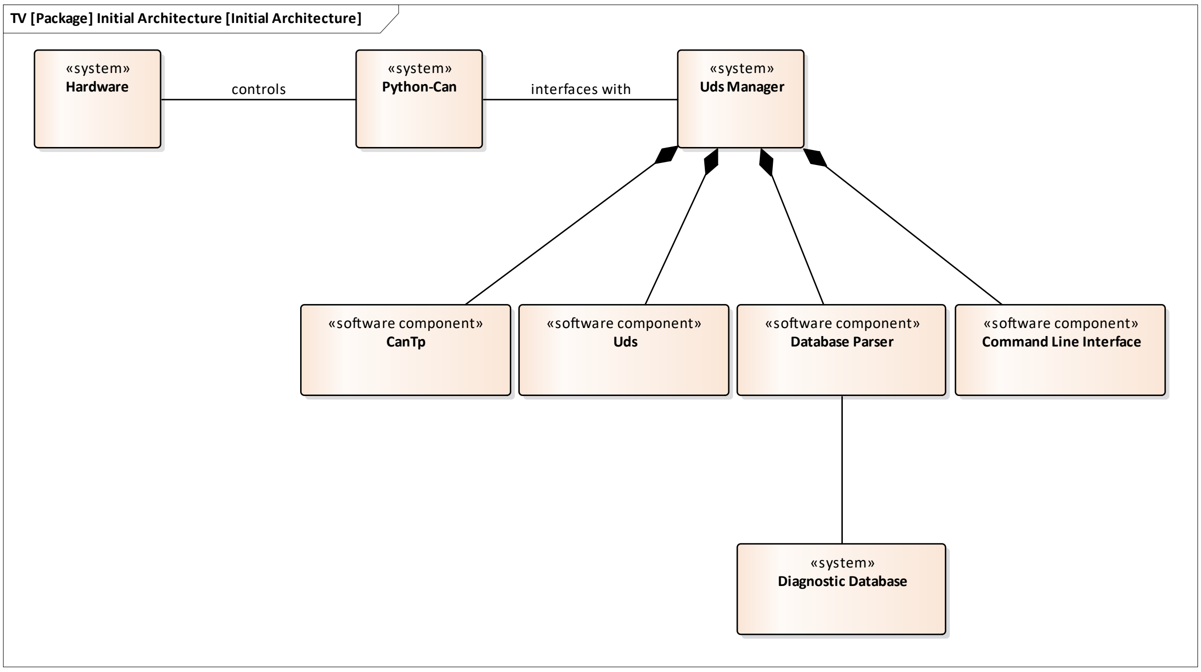


Figure 4 - Initial architecture based on system requirements

The architecture is based on the Model-View-Controller pattern, the command line interface being the View. The program would have been more monolithic with the Uds Manager owning all the components.

5.2. Prototyping

One main area of concern was the Database Parser. Due to the nature of the configuration databases it would be necessary to parse and create the class function content dynamically. Two methods were imagined:

1. Running the config tool and creating the relevant files. This is similar to how most of the C and C++ CAN signal auto-coding tools work. The configurations are parsed and the files are created at build-time behind the scenes. The advantages to this are that when the files are created the user will be able to easily see the interface and can use code-completion tools. The disadvantages are that if the files are not regenerated after changes in the database definition the program will know nothing of the new values.
2. Dynamically generating the code at runtime. The advantages of this are ease of use, however a significant drawback is the lack of transparency to the user. Most users rely on Integrated Development Environments (IDEs) and autocompletion tools for coding, and these would not work as there would be no source for the IDE to parse. It would be very easy for the user to enter a mis-spelled word and have the program produce exceptions or errors.

Some initial prototypes were performed to investigate the structure of the ODX and CDD files, and the ability of Python to parse these files. It was decided that as the ODX file format is defined by a standard and the CDD format is proprietary to Vector Informatik that the ODX would be the preferred format, and that a CDD parser would use Extensible Stylesheet Language Transformation (XSLT) to convert into an ODX, and then parse the ODX.

Method 2 was selected as the appropriate mechanism as it was able to parse and extract the necessary information fairly quickly from the initial tests.

5.3. Final Architecture

The final system architecture is detailed in Figure 5. This was chosen after further researching the solution space and analysing the system requirements.

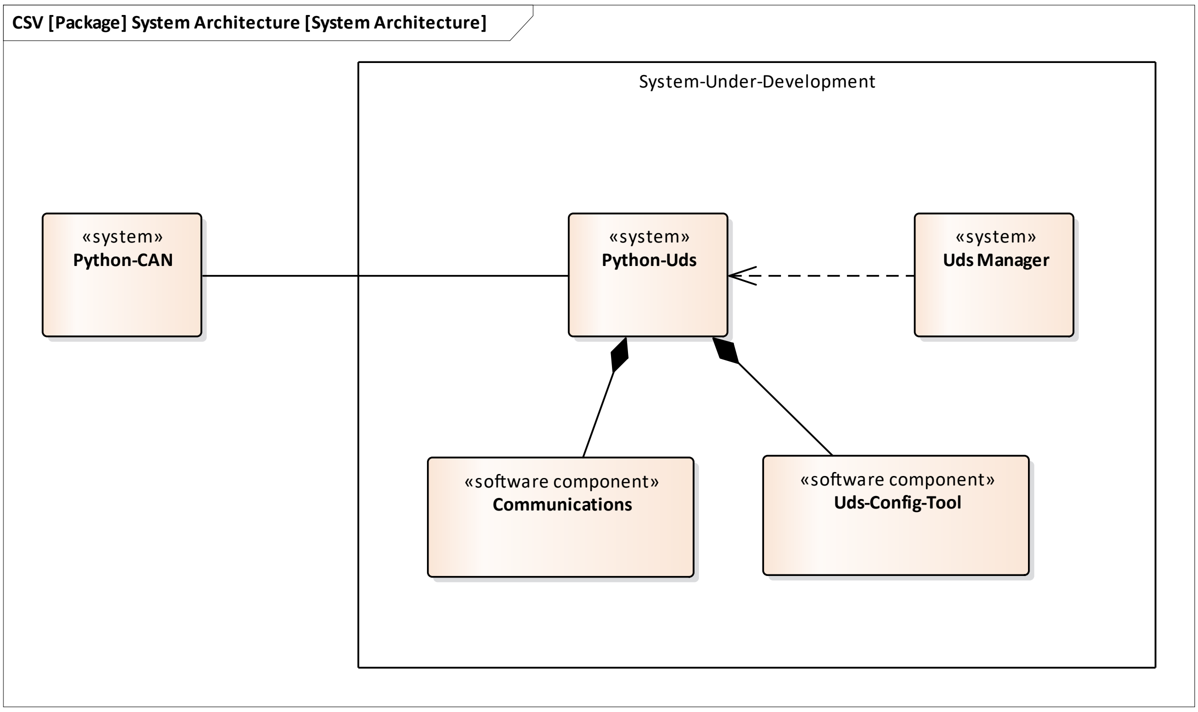


Figure 5 - Final architecture after further investigation and development

This architecture still follows the Model-View-Controller pattern, but separates out the components that are specific to Uds communication into a separate package which can be deployed and released separately to the Uds Manager.

The main change to separate out the Communications aspects of UDS and CanTp, and the ODX parsing and UDS protocol configuration into a separate system came from the need for re-use. Most of the use-cases involved users accessing the API and functions without using the command line tool, therefore splitting out the command line functionality into the UDS Manager system. The UDS Manager can simply use the published API and provide a command line interface to the user to perform their functions.

The UDS Manager system would be a more proprietary solution and would be retained by Embed rather than being released open source. Most of the requirements regarding the release of the code were based on the hobbyist use-cases so it did not seem to violate the intent of the requirements keeping this functionality closed-source.

5.3.1. Packages-to-Develop

From the final architecture a set of software components were selected for development.

5.3.1.1. Python-Uds (System)

The Python-Uds package will contain all the communications protocol handlers for UDS and any of the transport protocols to communicate with the vehicles. It also contains the configuration tools to parse the diagnostic databases. It contains two primary packages, Communications and Uds-Config-Tool. Communications handles all the physical communication, while Uds-Config-Tool handles the database parsing.

5.3.1.1.1. Communications (Software Component)

This component is the lower level handlers for communications. It will deal with establishing the communications channel, and the transmission and reception of data. The Communications system should be completely usable without the need for anything from the Uds-Config-Tool.

5.3.1.1.2. Uds-Config-Tool (Software Component)

This component handles the parsing and creation of the code necessary for the user to interact with the ECU at a high level. It will parse the diagnostic database and present the user with the exposed services and enumerations defined in the database. This component will contain the dynamic code generation functionality and will integrate with the Communications component.

5.3.1.2. Uds Manager (System)

This is the command line interface to the Python-Uds packages. It provides an interface to configure and set the parameters for the connections as well as specify the script file to be executed. This constitutes the view part of the model-view-controller pattern.

5.4. Off-The-Shelf Component Selection

Part of the requirements for this project is to use as many open-source projects to fulfil the project goals, primarily to lower cost and complexity of development. The following packages were selected as usable components for the system.

5.4.1. Python-CAN

This is an Open Source package developed by Dynamic Controls and has had several major contributors in the last 8 years. It provides an abstracted CAN interface for several popular hardware interfaces including Peak Systems and Vector Informatik which are two of the most popular CAN vehicle interfaces on the market. It also provides an interface to serial devices which should work with the common ELM327 CAN hardware, as well as the Bluetooth variants, and a socketcan interface for Linux.

This package is well documented and provides a good lower level interface and minimises the development effort for the Python-UDS system.

At the beginning of the project enquires were made to the current lead developer of python-can if they wanted the CAN Transport protocol implementation for their package. This would have split out the implementation but would have enhanced an already existing package allowing Embed to give back to the open source community. It was decided that this would not be suitable for their project model and they were not interested in integrating this feature.

5.4.2. Logging

Effective logging of programs is essential in fault finding, so early in the project a logging solution was required. Python provides a standard library for logging called ‘logging’. This provides a single package for multiple source files to integrate with, producing a coherent logging output. It could be loosely called a singleton pattern, but as the main methods are not part of a class and are global methods to the package, it could be argued that this does not follow the standard singleton pattern.

5.4.3. Configparser

Configparser provides an interface to configuration files using either .ini format or .json. It is a standard python library and is commonly used for configuring modules. This would also be useful to pass configuration parameters across modules (such as the required interface or the request and response IDs for the CAN Transport Protocol).

5.4.4. Argparse

Argparse is a common python library providing a parsing tool for command line parameters. This would be used in the Uds Manager package to allow the user to configure the communication channel and to set up the script to be executed.

6. Implementation

Initially the idea was to release a version 1 (V.1) release of the software which would support all of the initial system requirements, however during the implementation it became obvious that this was unachievable and the intention changed to releasing a Minimum Viable Product (MVP) which would include basic functionality and proof-of-concept code. This would be set as version 0.1 (V.0.1). A release schedule would be set up to enhance the functionality incrementally until the code reached the intention for V.1.

A more detailed breakdown of the classes, structure and behaviour is included in Appendix F.

6.1. Communications

UDS Communications were split into two sub-components, the UDS communication and the Transport Protocol.

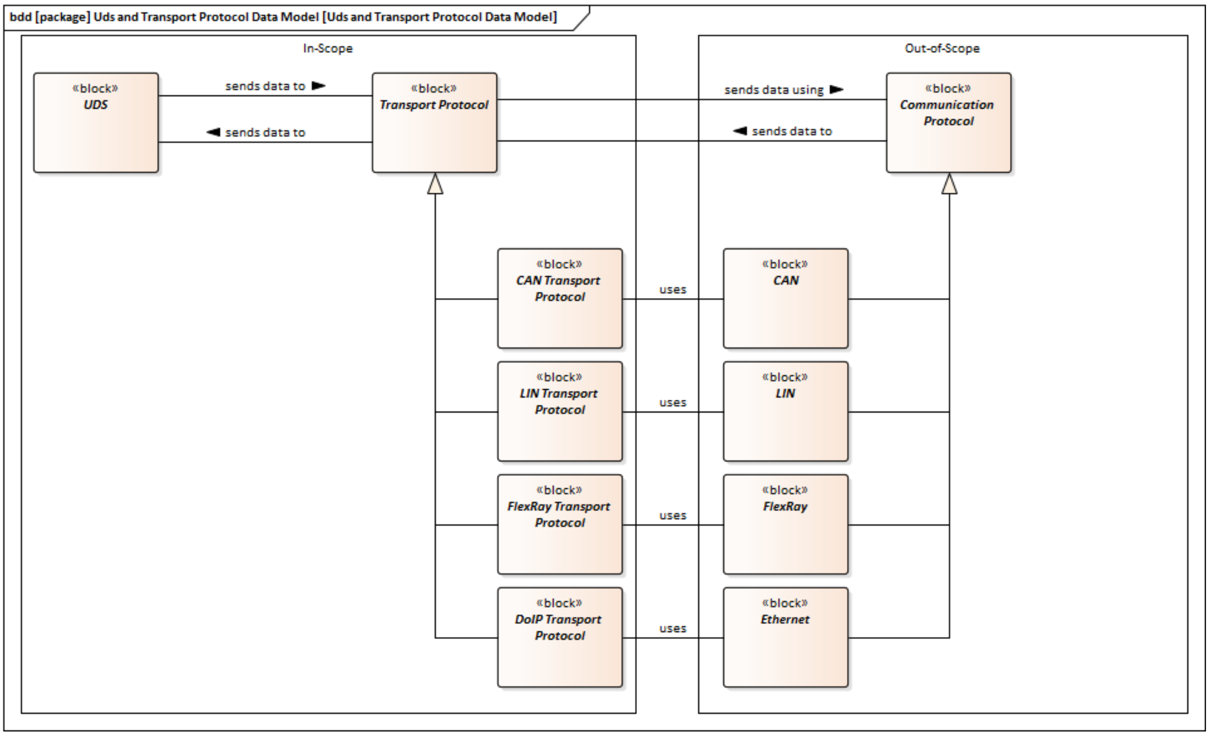

bdd [package] Uds and Transport Protocol Data Model [Uds and Transport Protocol Data Model]
In-Scope
Out-of-Scope
«block» UDS
sends data to ►
◄ sends data to
«block» Transport Protocol
sends data using ►
◄ sends data to
«block» Communication Protocol
«block» CAN Transport Protocol
uses
«block» CAN
«block» LIN Transport Protocol
uses
«block» LIN
«block» FlexRay Transport Protocol
uses
«block» FlexRay
«block» DoIP Transport Protocol
uses
«block» Ethernet

Figure 6 - Basic abstract model of the communications architecture

The communication architecture described in Figure 6 shows a basic description of the communications flow. UDS abstracted from the communications layer and does not care about the transport. There is a small amount of coupling between the UDS service and the transport as certain transport protocols place constraints on how large the transport frame can be (for CAN TP the maximum size is 4095 bytes as this is the largest number that can be stored in the data length (DLC) signal (ISO, 2013, page 19)

Initially the aim is to support CAN communications, the Communication Protocol side of the system is handled by the Python-CAN package, so the development is for the generic transport protocol interface and the CAN Transport Protocol.

The design decision was to create an interface definition for the Transport Protocol called iTp, which would be utilised by a Factory class to create the transport protocol instance. Each of the transport protocol implementations would follow the iTp interface so that the UDS instance is abstracted from the implementation of each of the TP classes, this is described in Figure 7.

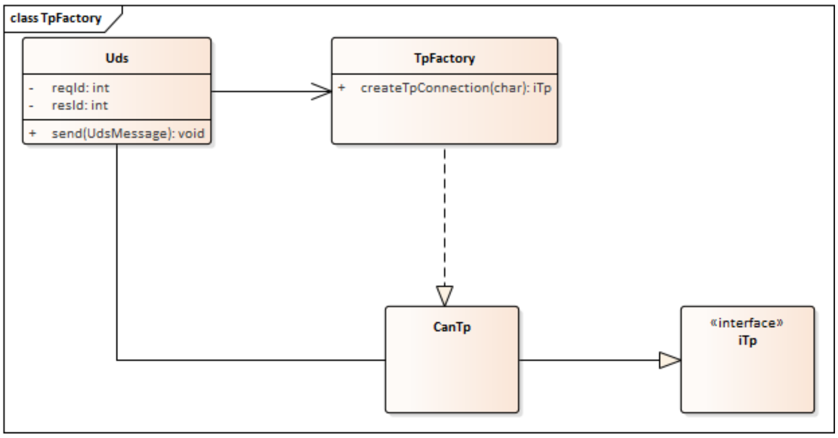


Figure 7 - CanTp class created from TpFactory

Going deeper into the *CanTp* class, its main dependency is the python-can package. Python-can has a well-defined and abstracted interface for the usage of its *Bus* objects. Each supports the same *Bus* interface, but the constructor for each of interfaces is different.

```
PcanBus(channel,
        state=BusState.ACTIVE,
        *args, **kwargs)

VectorBus(channel,
          can_filters=None,
          poll_interval=0.01,
          receive_own_messages=False,
          bitrate=None,
          rx_queue_size=16384,
          app_name='CANalyzer',
          fd=False,
          data_bitrate=None,
          sjwAbr=2,
          tseg1Abr=6,
          tseg2Abr=3,
          sjwDbr=2,
          tseg1Dbr=6,
          tseg2Dbr=3,
          **config)
```

This obviously implies that the parameters needed for each of the instances can not be passed from the Uds object so there must be a configuration that can be seen globally for the connection.

The configparser package was used to create a global config which can be shared with the constructor for each of the connection constructors. This created the need for a default configuration which would have to be used in place of a user-specified configurations.

The relevant parameters were analysed and a set of configuration parameters were created, the defaultConfig.ini file is located in the *\uds_configuration* folder in the source directory. At present the configuration does not support the creation of multiple ECU connections, but has been scheduled for the V.0.2 release.

6.1.1. Python timing problems

During the development of the CanTp class, it was found that implementing timing constraints in Python proved difficult.

Most of the CanTp behaviour needs to be non-blocking as it must be monitoring for flow control messages during the sending function this means that the basic *sleep*[3] function is unsuitable. The ResettableTimer class was developed to provide non-blocking timer functionality but unfortunately during testing it was found that the resolution of this timer could not improve past approximately 0.015 s, and the accuracy was anywhere from a few percent out to several hundred percent, sometimes exceeding 0.7 s which starts getting close to the default 1 s timeout behaviour.

When the timer was not implemented, the code would execute incredibly fast, completing a maximum size payload transmission in less than 0.010 s when using the virtual bus. However, this is not possible on an actual CAN bus as the transmission of the CAN messages can not be completed that quickly.

It was found that the standard python *time* method's resolution varies depending on the operating system. Windows standard timer resolution Is around 16ms which accounts for the instability and the resolution limit (it was later found that the standard windows timer has a 60 Hz frequency). After moving to the *perf_counter*[4] method also included in the time module provides greater accuracy, and testing this in isolation on the ResettableTimer class allowed the necessary millisecond resolution. This did not completely solve the issues, but they are significantly improved.

Two further attempts to improve the accuracy were conducted:

1. The first was to set up a class which inherited the *threading.Thread*[5]*.* This thread would execute in parallel with the main application and when its *expired* method was executed would immediately return the result without having to calculate the elapsed time, however this proved just as inaccurate.
2. The second was similar to the threading test, but used the multiprocessing library. This was very difficult to code and at present does not seem to work effectively. (This may be due to the authors lack of understanding on communicating data across processes). Details on the results can be seen in Appendix H.

After these investigations it was decided that another avenue of investigation would be to try using an external library written in C or C++. Python has a very mature and well-developed package ctypes which provides access to static or dynamic compiled libraries. This would be intended for after the V.1 release as an enhancement unless it proves to be limiting the functionality of the software.

6.2. Uds-Config-Tool

6.2.1. Dynamic Code Generation

There were several hurdles for dynamically generating the code necessary to create the desired user interface. One large hurdle with dynamic code generation is testing. Most code can be checked against unit tests to verify its functionality, however when the code is dynamically generated this is not as easy. One approach is to break the dynamically generated functionality into axioms which can be verified against a set of unit tests, when these are later built into a larger function a "chain-of-trust" is created which helps to add confidence that the solution is valid.

[3] The inbuilt python module *time* has a method *sleep* which pauses the execution for a set period of time. This is a blocking behaviour and would stop the program execution. This means that it would be unable to monitor for any flow control messages for that period.

[4] Perf_counter is a high accuracy measurement of the time since the beginning of the method call.

[5] Threading is the main python library for multithreaded applications.

An analysis was performed to find the set of operations that are likely to be performed on the returned data from the server, a series of basic operations were selected and added to DecodeFunctions.py. A set of tests were written to verify these functions, the subsequent methods in ReadDataByIdentifierMethodFactory.py use these functions to build up the parameter code. An in-depth example is included in Appendix I

After the initial investigation into parsing the ODX file to create the necessary functions was performed this part of the project was left until the primary communications interface was developed. After the first development of the code to parse the ODX file and create the necessary functions was completed, it was found that the conceptual communications model was not immediately achievable.

The aim was that the user would interface with the Uds connection in the following syntax:

```
        Ecu.readDataByIdentifier(‘ECU Serial Number’)
```

Where Ecu is the instance of the Uds class.

It was possible to allocate a method to an existing class, but the method would have to contain all of the necessary references to the supported sub-functions for that service. The following UML diagram defines the object inheritance and allocation.

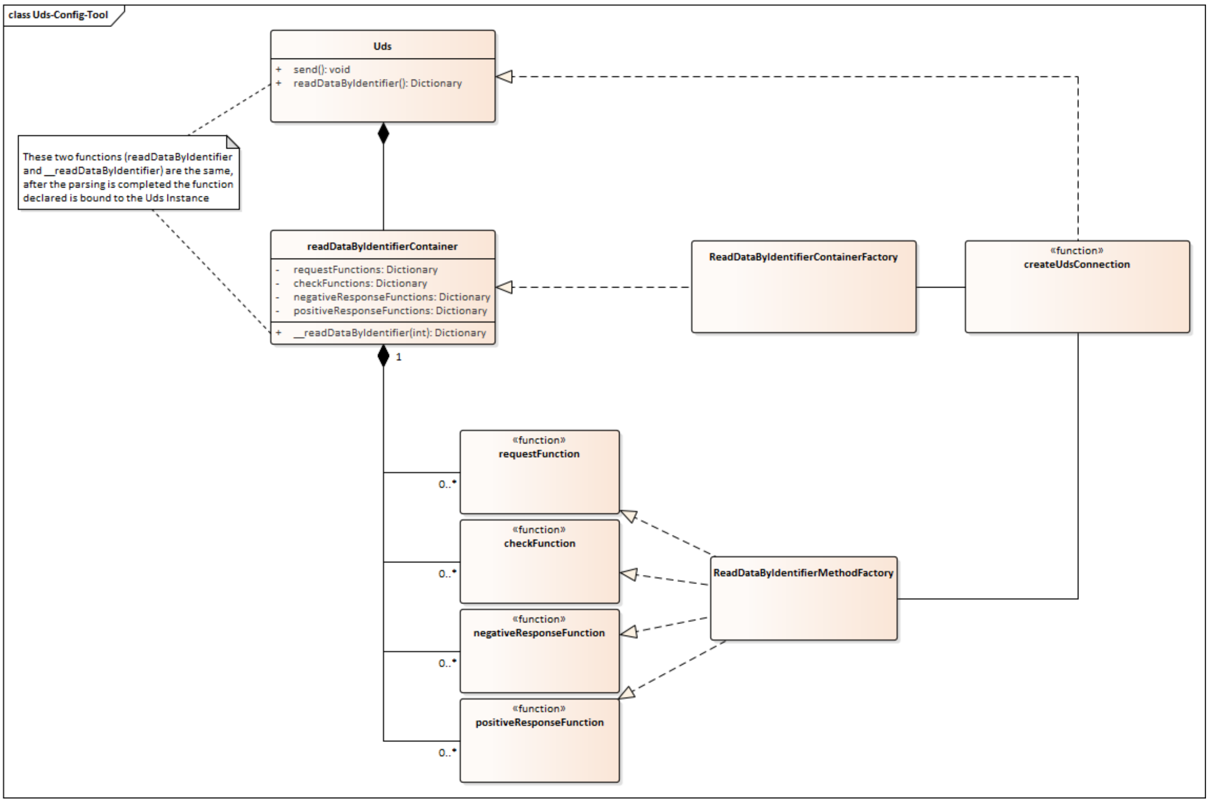

class Uds-Config-Tool
Uds
+ send(): void
+ readDataByIdentifier(): Dictionary
These two functions (readDataByIdentifier and __readDataByIdentifier) are the same, after the parsing is completed the function declared is bound to the Uds Instance
readDataByIdentifierContainer
- requestFunctions: Dictionary
- checkFunctions: Dictionary
- negativeResponseFunctions: Dictionary
- positiveResponseFunctions: Dictionary
+ __readDataByIdentifier(int): Dictionary
1
ReadDataByIdentifierContainerFactory
«function»
createUdsConnection
«function»
requestFunction
0..*
«function»
checkFunction
0..*
«function»
negativeResponseFunction
0..*
«function»
positiveResponseFunction
0..*
ReadDataByIdentifierMethodFactory

Figure 8 - Relationships between the service classes when created by the UDS Config Tool[6]

Each of the services has the same basic template of operations, but the specific behaviour executed differs depending on the service. A breakdown of this can be seen in Appendix F.3.2 and F.3.3.

[6] Currently UML does not support the modelling of functions as objects. The Class objects have been given the stereotype ‘Function’ to assist in clarifying the diagram.

When *createUdsConnection* is called it parses the database definition and creates a set of Service Container classes (one for each service found in the file) which all inherit the *iContainer* class. As it finds relevant sub-function definitions it parses the data structures and creates the relevant functions, these can be; Request, Check, Negative Response and Positive Response. These functions are given a key and added to the appropriate dictionary in the Container object. Once the file has been parsed the container is set as an attribute of the Uds object and a static method in the container is bound to the class. By doing this the container class defines the behaviour for the service, and provides the interface with the relevant parameters, and the Uds class has no dependency on the service so is completely decoupled.

Comments were made during informal code reviews that this is an unnecessary step, and simply having a set of defined functions that are part of the Uds class definition (similar to the udsoncan project (Lessard, Pier-Yves, 2018)) would be simpler, easier to test and possibly less error prone. These arguments are valid and may provide clearer interfaces but the current solution provides the desired functionality and no further changes will be made for the V.0.1 release.

6.3. Review

As with all code produced at Embed, code reviews are required to insure quality. As this project is assessed it was decided that reviews should only highlight failings in the code and no other user intervention would be allowed before the Final report was submitted.

This review was performed quite late in the project and only one person was able to provide any feedback on the developed code and the review comments have been added as issues to the GitHub repository. As the software currently communicates with a real ECU and provides the necessary base functionality to move forward, the aim is to set out a series of simpler sub-releases on V.0.1 which incrementally add the necessary functionality until a more mature system is produced, at which point an official V.1.0 release will be made and the old tools will be phased out. Regular reviews will be scheduled before the release of each new version. This plan has been put forward to the project management at Embed and they are satisfied with that goal.

7. Validation & Verification

Initially the plan was to develop the tests from the system requirements after the requirements phase. This plan failed as during the design and implementation phases the initial concepts for the code proved more difficult than expected and this phase was dropped as it was clear that the V.1 release was not achievable.

The intent going forward is to still create the tests for the requirements before the V.1 release to fulfil the project goals, the main reason for this is that it is good practice.

A part of this project was to use TDD during the implementation phase to develop the system and provide as much unit test coverage as possible. This proved to be quite difficult initially as the main system relies heavily on the python-can package for the interface and this dependency would have to be satisfied for correct unit testing.

Python-can provides a virtual interface for testing, and in the early stages of the project this was used to prove out concepts and initial tests. This interface is useful for the integration testing as this system relies on hardware for its interface it would mean that ideally a hardware test setup would have to be available for consistent testing. While this has been assembled for physical integration tests it should not be essential.

The python unit testing patch functionality was only discovered late in the implementation phase which provides the needed functionality. It allows specified function calls to be overridden with custom methods or return values to “mock” their real functionality. This behaviour allows each unit to be tested and its interface simulated to remove any dependencies from the unit tests. After this discovery a more comprehensive unit test suite was developed. Most of the code in the uds_communications module is covered by unit testing and some of the uds_config_tool module is as well, namely the decode functions which are detailed in Section 7.2.

The aim for V.1 is to have all of the code unit tested, and for there to be integration tests for both virtual bus connections and physical ECU tests against a test ECU. Currently the Uds-Config-Tool code has only been tested against an E400 using a standard bootloader and it was able to communicate with all of the implemented Read Data by Identifier services, and this was an informal test case primarily to validate the Peak CAN and Vector hardware interfaces and so is not detailed in this report.

7.1. Resettable Timer

The resettable timer was specified in a state machine format to make understanding the behaviour easier. A pitfall when looking at systems specified as state machines is to code them as state machines, despite it not being necessary.

The initial code created was programmed as a state machine, however after consideration it was refactored to make the logic simpler and more concise.

The unit tests are specified in such a way to exercise each of the transitions, the variables used to determine the state, and the method calls exercise those specified in the iResettableTimer interface, so if another implementation is developed or the code is refactored then it will be easy to run regression tests on the changes.

7.2. Decode Functions

The decode functions were another good example of TDD for verification and profiling for validation. Initially the ODX format was studied to further understand what functions would be required. A set of method signatures were created and a series of tests were written to exercise the needs of those methods.

The code was then developed until each of the tests passed, and at the same time some experiments were performed looking into better methods of implementing the functions. The experiment was to investigate the difference between normal structured programming, recursive programming or using the functools library included in Python which provides the *map*, *filter* and *reduce* methods.

These experiments are expanded on further in Appendix G, and the methods were exercised against profiling code which was able to determine which is the most efficient.

The results were conclusively that the recursive method yielded the slowest code, while the structured and map/reduce code were comparably fast, but the structured code was easier to read. It was decided to use the map/reduce code however as it was an interesting learning experience and may prove useful to others.

8. Delivery

To deliver this project, GitHub was chosen as the repository for the source code, while readthedocs.io was chosen as the documentation and API reference.

Python has a public package index (PIP) which is used to distribute the code in a packaged format read to use. This was implanted as the deployment mechanism.

8.1. Documentation

Documentation for this project is dealt with in two parts; code and usage.

Code documentation attempted to follow the principals of self-documenting code where the functions and variables are declared in such a way that they provide their own documentation. The intent is to document all of the functions using the Doxygen markup so that the API documentation can easily be produced at each release. This is still in progress with most of the Uds and CanTp code marked up but some of the other modules are still in progress.

The general user documentation is to be presented using readthedocs.io which provides a hosting platform for documentation. This was chosen as it integrates well with GitHub and is quite popular and well known. It also provides the ability to link the documentation to the released version of the code.

8.2. Open Source Distribution for Python-Uds and Uds Config Tool

Python-Uds is hosted on a pubic GitHub repository which will allow users to download, modify and check in any code they modify. This is a very popular method for open-source projects. The Uds Config Tool is part of the Python-uds package and will be included with the package.

The package will also be made into a pip installer so that users can easily install the package onto their distribution using the python pip tool.

The Uds Manager package is Embed specific and will be kept in the company's SVN repository. There is an internal process for releasing the different versions and this will be followed to comply with the company standards. The Uds Manager package will detail its dependency for Python-Uds and the specific version required.

8.3. Public Links

8.3.1. Python-UDS GitHub Repository

The python-uds public repository is available at the following link

https://github.com/richClubb/python-uds

8.3.2. Python-UDS read-the-docs.io

The python-uds documentation is available at the following link

https://python-uds.readthedocs.io/en/latest/

8.3.3. Python-UDS PIP installation instructions

Python-uds requires Python 3.7 as the main python interpreter. To install Python-UDS open a command line and execute the following command (assuming that python 3.7 is the python interpreter on the PATH)

```
pip install python-uds
```

9. Maintenance

Initially this phase was to be ignored and dealt with internally at Embed after the completion of the TM 470 report. Due to the problem with the development this was re-considered and as the project is being released as V.0.1 with incremental releases leading up to V.1 maintenance becomes more important.

As mentioned earlier in the report the intent is that there should be a scheduled release every month to enhance the functionality until all of the system requirements are met. At which point V.1 will be released and the maintenance cycle will be re-considered.

The currently known bugs and enhancements are logged on the GitHub project page and this will work as the project management page going forward.

Eric Raymond expands on the nature of the maintenance and release cycle of open source software (Raymond, 2001) The conclusion is that to get people interested in contributing and using an open-source tool the release cycles have to be short and succinct. This matches general agile programming philosophies of short development cycles. (Farley, 2016)

10. Summary

Each sub-section goes into detail on a specific part of the project that requires further investigation, there is a final summary at the end of the section.

The code has been released as V.01, and is available on the links in Section 8.3. Although the intention at the start of the project was for a more fully featured release, the current release is a significant step towards achieving that goal.

10.1. Uds Communication & Can TP Development

After further discussion and investigation, it may have been better to model the CAN stack and communications architecture on the AUTOSAR block architecture. This provides a very organised approach to the communications structure, transmission and processing. The main difference in to the current architecture is the inclusion of the PDU router (PduR) which acts to link the higher-level components to the lower level signals. By adopting a PduR it would have allowed the higher-level objects to use both UDS and traditional communications side-by-side, so if a parser for the DBC or similar had been created it could have brought the signals on CAN up to an ECU object for access.

The AUTOSAR standard also defines the interface layers between the different blocks so it would have been easier to copy their standard interface rather than having to create a proprietary one.

The creation of the CanTp code was more complicated than anticipated, and has slowed down the project significantly. The management at Embed were not worried about the delay and are more interested in getting a well-constructed and maintainable project than having one delivered to the TM470 project deadlines. As their previous attempt failed due to lack of extensibility they are careful not to make the same mistake twice.

10.2. Project Lifecycle

The initial thoughts for this project were to use a simple method for project management, namely the classic waterfall process as the domain was well understood and the project had quite defined goals. This proved to be an incorrect assessment and the project would have been better suited to a hybrid approach. The initial lifecycle failed due to the lack of time needed to exercise each of the stages to completion, a more agile approach would have been suitable. Domain Driven Design advocates a longer requirements and context analysis phase than other agile methods, with the intention to understand the user's needs and the wider system context, but does not place constraints on the rest of the development cycle and many adherents practice Agile lifecycles. This methodology fits well with the Embed's requirements capture process which is intended to satisfy ISO 26262 functional safety and Automotive SPICE traceability requirements. This may be a separate line of enquiry to enhance Embed's development processes for dealing with purely software-based projects.

After a review with employees of Embed it became obvious that a strict adherence to either agile or waterfall produced undesirable results. For example, a particular project was managed by an external company and followed a Scrum process. The main failing of this project was one of architecture. The final system was compromised as the different teams had developed solutions which were difficult to integrate, and as requirements changed over the course of the project the refactoring became a much larger part of the development process than the implementation of the desired new features. Similarly, a waterfall-based project encountered a similar problem was encountered as the project was near complete and was about to be deployed to a customer and a new requirement set was elicited from a missed use-case, this set caused serious problem with the developed system and meant that the delivered product was compromised. Robert Martin expands on the nature of these problems (Martin, 2018) stating *"If you give me a program that does not work but is easy to change, then I can make it work, and keep it working as requirements change. Therefore, the program will remain continually useful"*. Adherence to this concept is a good philosophy and regardless of the lifecycle used this should always be kept in mind.

These examples have been experienced several times by the reviewers at Embed. While the results are anecdotal and the exact reasons for these problems are not known the general consensus was that an ideal approach would be to perform a rigorous requirements elicitation and design phase (such as the ACRE process) to develop the architecture of the system, and then split the rest of the project into smaller "deep-dives" into the implementation with successive deliveries to the customer to provide validation reviews.

The reviews were happy with the use of TDD at the implementation phase and the concept of performing high-level TDD was seen as unanimously popular idea. It was decided that a new approach would be investigated to write and review tests on the high-level requirements before the design process begins. This approach would gain the benefits from greater understanding that comes from developing the tests on the requirements, however the tests may not be able to be exercised for quite some time as the "deep-dives" may not produce the functionality to later in the development, but regular regression testing should be performed at each development cycle to validate the developed functionality.

Despite the failure for this project, it has provided some insight into new methods and procedures which can be investigated for the company.

10.3. Overestimation of Complexity

The project as a whole was underestimated in both complexity and scope. The initial requirements and architecture discussions were completed quite early on and it was assumed that this would be a fairly simple project with the most complex part being the creation of the ODX parser and Uds-Config-Tool component. The CanTp communications was more complicated than expected as some of the behaviour was not as well understood and this led to a few failed implementations which became cumbersome and impossible to refactor when problems were encountered.

Initially after a detailed reading of the ISO 15765 standard a set of classes were developed as shown in Figure 9.

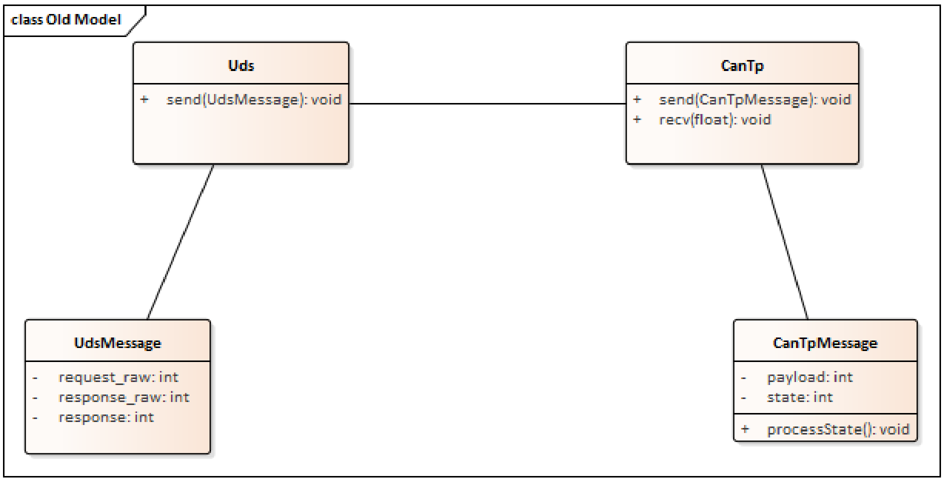


Figure 9 - Old model of classes used for UDS and CAN TP communication

The idea was that the user would create a Uds Message object, and that would contain the necessary functionality to format the message as per the service definition of the message (defined in the ISO 14229 standard), however this became very cumbersome and didn't fit the high-level interface described in Section 3.1. The UdsMessage class was dropped early on in favour of using a list data structure to represent the UDS PDU.

The CanTp and CanTpMessage were even more complicated with two separate state machines being specified for the class behaviour, the CanTpMessage state machine is shown in Figure 10.

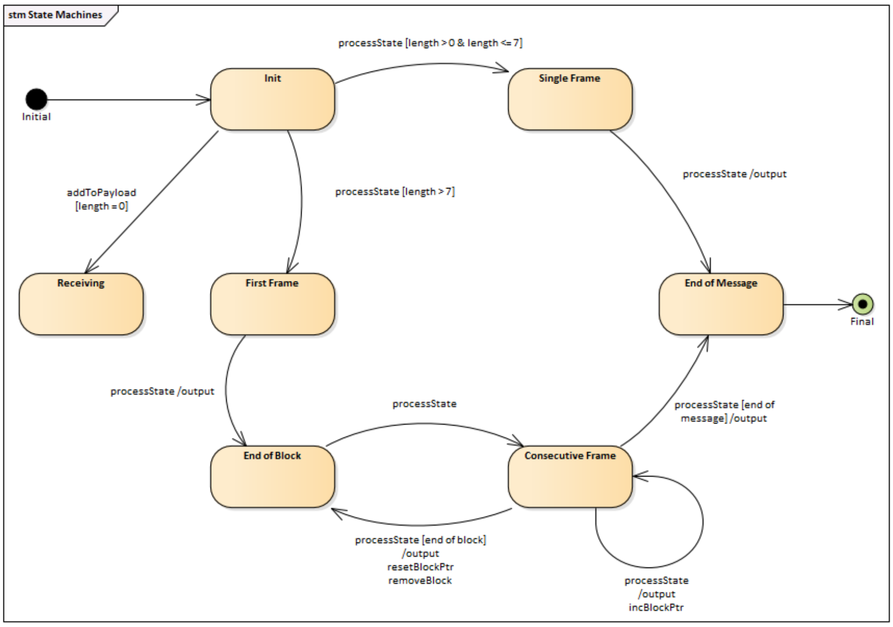


Figure 10 - Initial CanTpMessage state machine

This state machine is needlessly complicated and features states for both the transmission and reception of the multi-frame CAN TP PDUs, which are mutually exclusive events. Its intention was that it would remove some of the complexity from the CanTp class, but the interaction between the two different state machines became increasingly complex. It was simplified into the two separate state machines defined in Appendix F.2.1.2.2.1 and F.2.1.2.2.2

10.4. Project Management

The project management for this project has been a constant struggle, and part of this was to do with the underestimation of the complexity of the problem. It was assumed early on that the code would be trivial to create and thus would not require detailed project management as this presents a significant time overhead.

On reflection, the use of the GitHub project tools and systematic ticketing of both the tasks and milestones would have aided this considerably. After the first few problems the project should have switched to a more rigorously task-based approach.

Embed usually employs this approach using Trac (https://trac.edgewall.org/) and it works well for projects that involve multiple members of staff, but does add a significant time overhead as the tickets need to be maintained constantly to get the most from the process.

Along with accurate tracking of the progress of the project, a larger number of formal reviews would have been advised, both by members of Embed and by the TM 470 tutor. This would have assisted highlighting technical and project problems and may have resolved in being able to apply corrective action earlier in the project.

The project shall be using this process going forward using the GitHub project management to track its progress, tasks and bugs.

A more comprehensive project log would have been a good tool to show the different approaches to the problem encountered in the project, this would aid in justifying time spent solving issues in the project in general.

10.5. In Conclusion

The project has been challenging and has raised a number of interesting points at every part of the lifecycle. The principal failures are the underestimation of the complexity of both the CanTp module and the parsing of the ODX files to generate the UDS service code. Although it is disappointing that the software has not been developed to the desired level, there is a clear path going forward which should quite soon result in a useful tool.

A lot has been learned about Python in general, and some of the tools that it provides, especially the use of map/reduce/filter to work on iterable objects and more functional programming techniques. The major take-aways are to do with the management of a more complicated project, and how to handle difficulties during development.

As this was principally an implementation project, it was difficult to find literature to guide the development process. The project was driven principally by a number of industry text books (Holt / Perry, 2013), (Hull / Jackson / Dick, 2005), (Martin, 2018). Although this is likely due to the authors lack of experience, and greater communication with their tutor may have been able to guide towards better sources of technical literature.

The authors intent going forward is to spend more time looking over the exact failings encountered in this project and to structure a plan to address each of the shortcomings, both personal and professional so that they can improve their day-to-day work. A debriefing on this project has been scheduled for the end of September to feed back to the management and technical leads at Embed regarding this report and the state of the project at V.0.1. The intention is that they may also have feedback which will be integrated into the plan.

11. References

Lessard, Pier-Yves (2018) *udsoncan documentation* [Online] Available at: https://udsoncan.readthedocs.io/en/latest/index.html (Accessed 15 September 2018)

Linux Kernel (n.d.) *socketcan documentation* [online] Available at: https://www.kernel.org/doc/Documentation/networking/can.txt (Accessed 15 September 2018)

International Organisation for Standardization (2013) *ISO 14229-2:2013 Road Vehicles - Unified Diagnostic Services (UDS) – Part 2: Session Layer Services*, ISO Publishing

International Organisation for Standardization (2016) *ISO 15765-2:2016 Diagnostic Communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services*, ISO Publishing

International Organisation for Standardization (1994) *ISO 5725-1:1994 Accuracy (trueness and position) of measurement methods and results – Part 1: General Principals and definitions* [Online] Available at: https://www.iso.org/obp/ui/#iso:std:iso:5725:-1:ed-1:v1:en

Royce, Dr. Winston W (1970) *Managing the Development of Large Software Systems* [Online] Available at: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf (Accessed 11 September 2018)

Hull, Elizabeth / Jackson, Ken / Dick, Jeremy (2005) *Requirements Engineering*, Springer

Almeida, Daniel, A / Wilson, Greg / Hoye Mike (2017) *Do Software Developers Understand Open Source Licences?* [Online]. Available at: http://www.cs.ubc.ca/~murphy/papers/licensing/software-licensing.pdf (Accessed 15 June 2018)

Broy, Manfred / Gleircher, Mario / Kluge, Peter / Krenzer, Wolfgang / Merenda, Stefano / Wild, Doris (2009) Automotive Architecture Framework: Towards a Holistic and Standardised System Architecture Description. [Online] Available at: ftp://ftp.software.ibm.com/software/plm/resources/AAF_TUM_TRI0915.pdf

Hackaday (2018) *Hackaday Website* [Online]. Available at: https://hackaday.com/ (Accessed 15 July 2018)

Anool Mahidharia (2017) *Reverse-Engineering the Peugeot 207's CAN Bus* [Online] Available at: https://hackaday.com/2017/05/04/reverse-engineering-the-peugeot-207s-can-bus/ (Accessed 15 July 2018)

AUTOSAR Consortium (2017) *AUTOSAR Layered Software Architecture* [Online] Available at: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (Accessed January 2018)

Martin, Robert C. (2018) *Clean Architecture*, Prentice Hall

Raymond, Eric S. (2001) *The Cathedral & the Bazaar, Revised Edition*, O'Riley

Farley, David (2016) *Doing continuous delivery? Focus first on reducing release cycle times* [Online] Available at: https://techbeacon.com/doing-continuous-delivery-focus-first-reducing-release-cycle-times (Accessed 24 June 2018)

Holt, Jon/ Perry, Simon (2013) *SysML for Systems Engineering, 2nd Edition: A Model-Based Approach,* IET Publishing

Nuseibeh, Bashar (2001) *Weaving the Software Development Process Between Requirements and Architectures* [Online] Available at: https://ieeexplore-ieee-org.libezproxy.open.ac.uk/document/910904/

(Accessed 5 May 2018)

Osherove, Roy (2005), *Employing TDD and unit testing with Waterfall methodologies,* [Online] Available at: http://osherove.com/blog/2005/2/21/employing-tdd-and-unit-testing-with-waterfall-methodologies.html

12. Glossary

AFR – Air-fuel Ratio

ASAM - Association for Standardisation of Automation and Measuring Systems

AUTOSAR – An automotive standard to promote open software architectures for vehicle ECUs

CAN – Communication Area Network, a communication protocol used heavily in production vehicles. Used in almost all vehicle modules for inter-module communication

CanTp – CAN Transport Protocol

DBC – File format used for CAN message and signal definition

DDD – Domain Driven Design

DID – Diagnostic Identifier

ECM – Engine Control Module, sometimes referred to as PCM

ECU – Electronic Control Unit, a generic term for a vehicle control unit.

FlexRay – A communication protocol used in production vehicles. It is a high-bandwidth, time-sensitive protocol which features in-build redundancy mechanisms. Intended for use with safety-critical systems such as steering and braking.

Gordon – Embed's current tool for reprogramming ECUs

ISO – International Organization for Standardization

LDF – LIN Definition File, a file format used for LIN frame and signal definition

LIN – Local Interconnect Network, a communication protocol used heavily in production vehicles, mostly for lower functionality nodes such as door switches or single motors

OO – Object Oriented

PCM – Powertrain Control Module, sometimes referred to as ECM.

PDU – Protocol Data Unit, a generic term used to describe the packets of data transmitted by different protocols. E.g. a UDS PDU is different from a CAN TP PDU and a CAN PDU

PduR – PDU Router, an AUTOSAR component for routing signal and message information in the basic software layer.

RDBI – Read Data by Identifier

TCM – Transmission Control Module

TDD – Test Driven Development

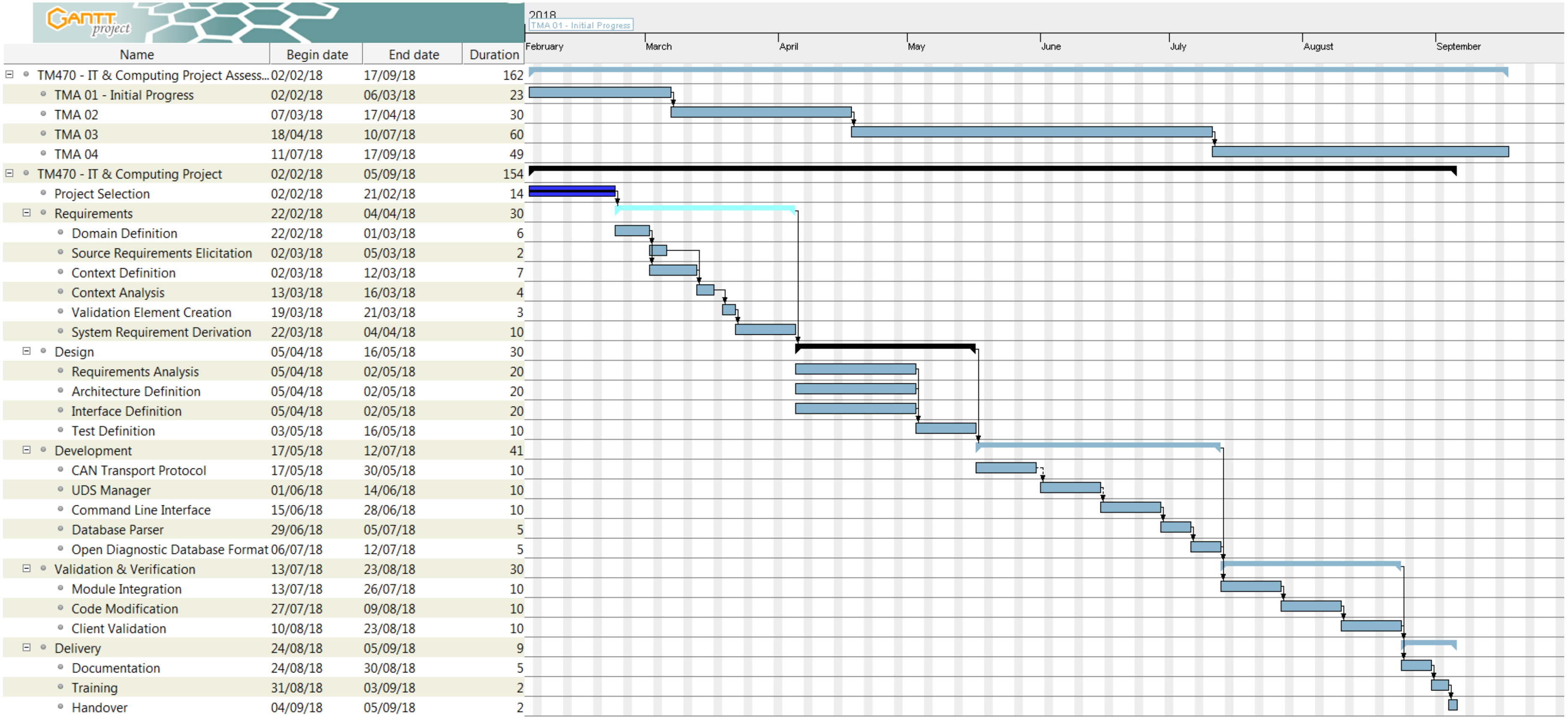
UDS – Unified Diagnostic Services

V&V – Validation and Verification

WDBI – Write Data by Identifier

Appendices

Appendix A – Project Gantt Chart



Name	Begin date	End date	Duration
⊟ ● TM470 - IT & Computing Project Assess...	02/02/18	17/09/18	162
● TMA 01 - Initial Progress	02/02/18	06/03/18	23
● TMA 02	07/03/18	17/04/18	30
● TMA 03	18/04/18	10/07/18	60
● TMA 04	11/07/18	17/09/18	49
⊟ ● TM470 - IT & Computing Project	02/02/18	05/09/18	154
● Project Selection	02/02/18	21/02/18	14
⊟ ● Requirements	22/02/18	04/04/18	30
● Domain Definition	22/02/18	01/03/18	6
● Source Requirements Elicitation	02/03/18	05/03/18	2
● Context Definition	02/03/18	12/03/18	7
● Context Analysis	13/03/18	16/03/18	4
● Validation Element Creation	19/03/18	21/03/18	3
● System Requirement Derivation	22/03/18	04/04/18	10
⊟ ● Design	05/04/18	16/05/18	30
● Requirements Analysis	05/04/18	02/05/18	20
● Architecture Definition	05/04/18	02/05/18	20
● Interface Definition	05/04/18	02/05/18	20
● Test Definition	03/05/18	16/05/18	10
⊟ ● Development	17/05/18	12/07/18	41
● CAN Transport Protocol	17/05/18	30/05/18	10
● UDS Manager	01/06/18	14/06/18	10
● Command Line Interface	15/06/18	28/06/18	10
● Database Parser	29/06/18	05/07/18	5
● Open Diagnostic Database Format	06/07/18	12/07/18	5
⊟ ● Validation & Verification	13/07/18	23/08/18	30
● Module Integration	13/07/18	26/07/18	10
● Code Modification	27/07/18	09/08/18	10
● Client Validation	10/08/18	23/08/18	10
⊟ ● Delivery	24/08/18	05/09/18	9
● Documentation	24/08/18	30/08/18	5
● Training	31/08/18	03/09/18	2
● Handover	04/09/18	05/09/18	2

Figure 11 - Gantt chart for the project

Appendix B – Project Log

Date	*Activity*	*Notes*
2018-3-13	Investigation	Investigation into python ctypes module
2018-3-20	Prototyping	Implementation of a ctypes interface for the Peak Systems PCAN-USB device driver. Initial investigation was successful. Further investigation into other device drivers was encouraged
2018-3-21	Prototyping	Implementation of a ctypes interface for the Vector Informatik VectorXL driver library. Initial investigation was successful but the implementation was more complicated than the PCAN-USB device driver. Research into other methods and mechanism for device support was scheduled
2018-4-1	Prototyping	Prototyped the use of dynamically defined functions in python to solve the problem of adding the sub-functions from the diagnostic database to the service classes.
2018-4-5	Investigation	Investigated python-can module to add support for multiple devices. The module looked very promising as it contained support for all the necessary devices as well as support for socket-can which was encouraging for cross-platform support
2018-4-7	Reading	Read "Requirements Engineering" by Elizabeth Hull, Ken Jackson and Jeremy Dick. The book covers the development, elicitation and derivation of requirements.
2018-4-16	Investigation	Investigated python transitions module for implementations of state machines.
2018-4-22	Prototyping	Prototyped CAN Transport Protocol implementation using python-can module. The results were promising using the internal loopback interface. Involved setting up threads to create listeners for the response message from the test ECU.
2018-5-5	Research	Vehicle interface research. During the investigation into the python-can module I found a very common CAN interface which utilises the ELM327 chipset. This is very cheap and a number of the devices on the market support WiFi and Bluetooth connectivity. This would be an ideal device to use but there is no direct driver support so this would be a separate task to extend the python-can device support. The ELM327 devices generally enumerate as a serial device and python-can supports CAN over serial so further investigation is required to see if this is a viable option
2018-6-1	Reading	Read "Clean Architecture" by Robert Martin – The book covers concepts for architecting software systems.
2018-6-5	Reading	Read "The Cathedral and the Bazaar" by Eric Raymond – Book covers in detail methodologies for managing open source projects and societal involvement in open projects.
2018-8-11	Research	Investigated the use of functional programming techniques to solve decoding UDS payloads.
2018-8-17	Research	Investigation into different non-blocking timer implementations to solve the "stMin" wait and "FC" wait timing problems.

Table 1 - Project log

Appendix C – Risk Assessment

Risk		Probability	Impact	Risk	Mitigation Plan
Impact of work / other commitments		High	Medium	Medium Risk	Put any non-essential project on hold until the end of the module. Plan in work to be done in advance to guarantee time. Discuss with employer the potential of using work-time to spend on TM470 project.
Complexity of project		Low	High	Medium Risk	Detail any potential areas for investigation and schedule in time for appropriate research.
Failure to meet TMA deadlines		Medium	Medium	Medium Risk	Put any non-essential project on hold until the end of the module. Plan in work to be done in advance to guarantee time. Discuss with employer the potential of using work-time to spend on TM470 project.
Failure to meet EMA deadline		Low	High	Medium Risk	No mitigation strategy.
Change of employer		Medium	Low	Low Risk	Standards are publicly available and no proprietary information is used in the project. Agreement in place with employer that material published is allowed to be released as open source.
Project scope too large		Medium	Medium	Medium Risk	Classify each task by its importance to the entire project, re-assess at regular intervals if all parts will be able to be delivered and re-prioritise.

Table 2 - Risk assessment table

Appendix D – High Level Requirements & Context Analysis

This section covers the initial requirements elicitation and analysis and includes the Goals, Capabilities, System and Stakeholder Contexts as defined in the ACRE process (Holt / Perry, 2012)

The content for sections D.1, D.4 and D.5 are generated from a SysML modelling tool. The tool currently has a bug where it is not possible to logically order the elements for export so the order is alphabetical. The author apologises for this.

D.1. Goals

These are the highest-level requirements defined by the customer and are intentionally broad. They describe the ultimate needs of the system and help to define the problem that needs to be solved.

<table>
<tr><td>Req: Goal:001</td><td>Communication Protocol Support</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Goal:
Support for UDS on multiple protocols, primarily CAN and LIN

Rationale:
Embed are expanding their UDS offerings and may require support for LIN in the future</td></tr>
</table>

<table>
<tr><td>Req: Goal:002</td><td>ECU Reprogramming</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Goal:
Reprogramming of ECUs

Rationale:
One of the main use-cases for the previous solution was the need to be able to program ECUs easily and cheaply without having expensive tools.</td></tr>
</table>

<table>
<tr><td>Req: Goal:003</td><td>Extensibility</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Goal:
The software should be easy to change and modify in future

Rationale:
The previous software solution contained all the configuration and behaviour in compiled code which made it very difficult to change. The most flexible part of the software was heavily constrained by the compiled code.</td></tr>
</table>

<table>
<tr><td>Req: Goal:004</td><td>Hardware Support</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Goal:
The software should be able to use multiple hardware interfaces, specifically Vector and Peak USB at a minimum.

Rationale:
Previous software solutions have been hardware limited and has meant that customers can not use the hardware they have available and must use specific tools. This has caused problems in some use-cases.</td></tr>
</table>

Req: Goal:005	UDS Support
Status:	Accepted
Goal: The system needs to provide an interface to access UDS implementations on ECUs Rationale: The purpose is to replace an existing tool, so the software should be able to communicate with the UDS implementations as defined by the diagnostic standards	

Req: Goal:006	Open Source Release
Status:	Accepted
Goal: The software should be released as open source where possible Rationale: The software is intended to be free so it makes sense to release it as open source to be able to get community involvement. It also means that clients can access and improve the software if they wish.	

Req: Goal:007	Technology Learning
Status:	Accepted
Goal: Use the project to learn about technologies and tools Rationale: The purpose of the TM 470 project is to learn so the software should provide ample opportunities to learn.	

D.2. Stakeholder Context

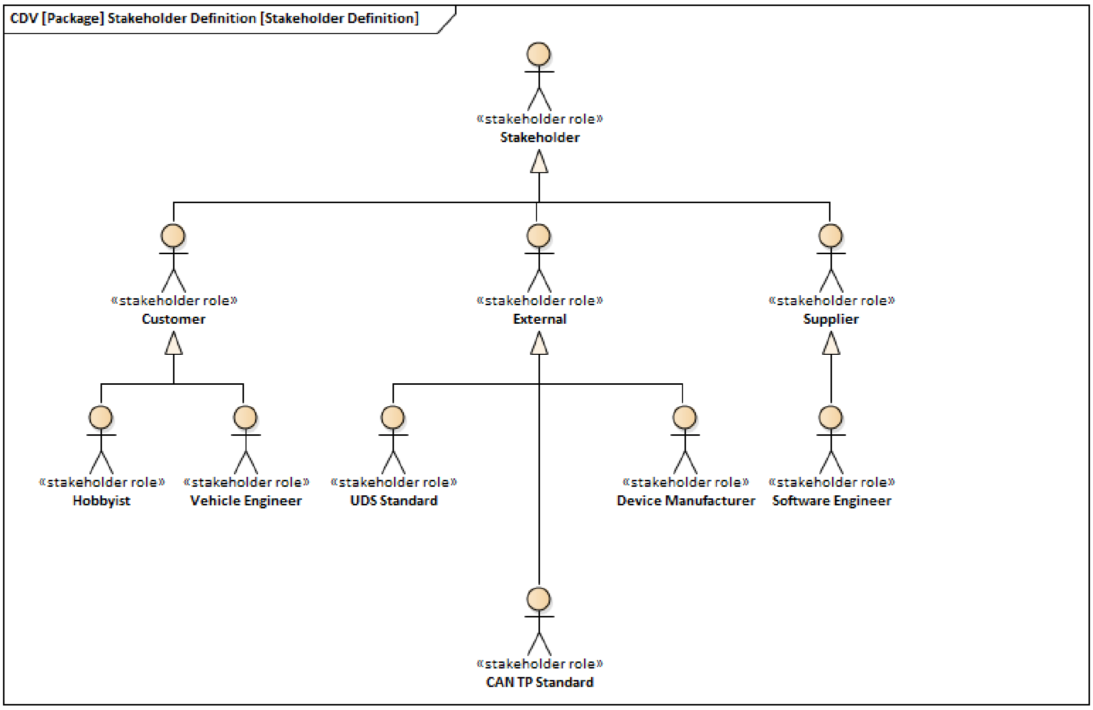


Figure 12 - Stakeholder context definition

The Customer, External and Supplier stakeholders are taken from the ACRE process (Holt / Perry, 2012).

D.2.1. Customer

The Customer stakeholders are the ultimate users of the system, they may directly or indirectly use the system but generally are flexible and can be negotiated with.

D.2.1.1. Hobbyist

The hobbyist is a person who works with vehicles for fun.

They have a varying level of technical skill, and may not have access to the "official" data sources and may be working with information they have collected or created themselves, and may be trying to exploit the vehicle in some way to gain functionality

D.2.1.2. Vehicle Engineer

A vehicle engineer is a professional and their job is to work with vehicles and ECUs.

They will generally have access to official information on the vehicles and ECUs on which they are working. It is likely that they will be working completely within legitimate means

D.2.2. External

The External stakeholders generally place constraints on the system and MUST be satisfied. They are generally entities such as standards bodies or government organisations who's needs are not negotiable.

The main contexts in this category were defined as the standards which govern the protocol definition, along with the manufacturers of the devices used to interface with the networks.

D.2.3. Supplier

The Supplier contexts are the entities involved in delivering the system. The needs of these contexts are generally the most flexible.

The only context for this project was the Software Engineer.

D.3. System Context

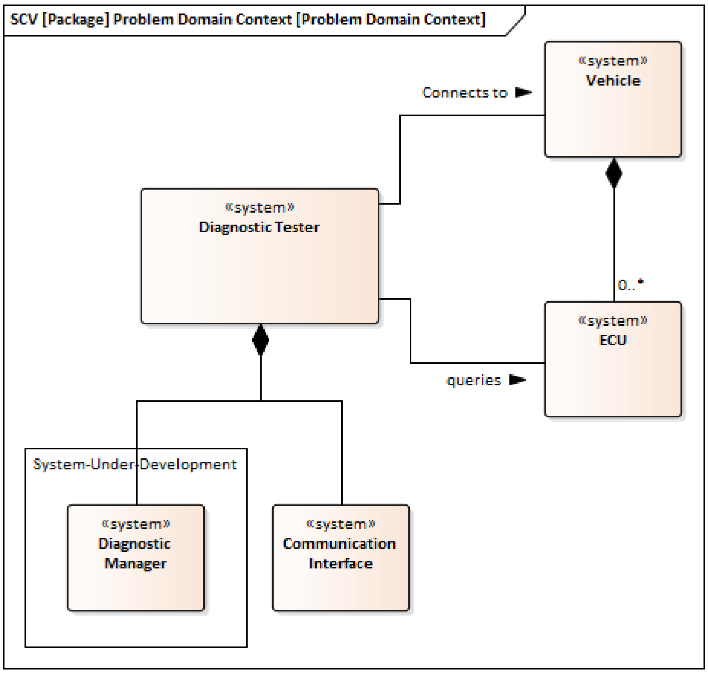


Figure 13 - System context definition

A Diagnostic Tester consists of a Communication Interface and the Diagnostic Manager, the Communication Interface is the hardware used to interface to the vehicle, and the Communication Interface is the software used to drive the Communication Interface.

A Vehicle generally consists of 0 or more ECUs each performing specific functions. Generally, the concept is that the Diagnostic Tester interfaces physically with the vehicle as a whole, and will perform communication or functions on multiple ECUs. Each of the ECUs have specific identifiers which allow concurrent communication and there are multiple addressing schemes to handle this behaviour.

D.4. Capabilities

The capabilities define what features are needed from the given system. They sit above the system requirements and should be phrased in an implementation agnostic way. The capabilities are analysed from the context of the system, systems or system-of-systems depending on the scope. In this case the system is primarily software with hardware interfaces so we are considering it from a single system context, the Diagnostic Manager.

<table>
<tr><td align="right">Req: Cap:001</td><td>Multiple Device Communication</td></tr>
<tr><td align="right">Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
It should be possible to perform communication between multiple devices simultaneously

Rationale:
For a hobbyist it may be necessary to get information from multiple devices at the same time.

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td align="right">Req: Cap:002</td><td>Open Source Code</td></tr>
<tr><td align="right">Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
Any software published should be released under an open source licence and distributed free-of-charge</td></tr>
</table>

<table>
<tr><td align="right">Req: Cap:003</td><td>Platform Support</td></tr>
<tr><td align="right">Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
There should be support for Linux and Windows operating systems

Rationale:
A large number of hobbyist users’ interface with vehicles using the Raspberry Pi or other Linux based operating systems, while most vehicle engineers tend to use Windows as most of the other development tools are Windows based</td></tr>
</table>

<table>
<tr><td align="right">Req: Cap:004</td><td>Protocol Support ISO 15765</td></tr>
<tr><td align="right">Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
The interface shall conform to the ISO 15765 standard

Rationale:
ISO 15765 defines an interface for transport protocols between UDS and the physical layer communication protocols, such as DoIP, CAN and LIN.</td></tr>
</table>

<table>
<tr><td>Req: Cap:005</td><td>Published API</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
There should be a documented API to interface with

Rationale:
The user needs to be able to interface with the system and they may not be expert users and require some explanation on the behaviour</td></tr>
</table>

<table>
<tr><td>Req: Cap:006</td><td>Support for ISO 14229 Services</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
Support for all ISO 14229 Services

Rationale:
To correctly support UDS conformance to the ISO 14229 standard is needed</td></tr>
</table>

<table>
<tr><td>Req: Cap:007</td><td>Support for OBD protocol</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
It should support the OBD-II protocol modes

Rationale:
For a hobbyist the use of the OBD-II modes is a useful addition.</td></tr>
</table>

<table>
<tr><td>Req: Cap:008</td><td>Support ISO 14229 Communications</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Capability:
There shall be support for communication over UDS as defined by ISO 14229</td></tr>
</table>

D.5. System Requirements

This section lists all of the system requirements for the Diagnostic Manager.

<table>
<tr><td>Req: Sys:013</td><td>ECU Reflash</td></tr>
<tr><td>Status:</td><td>Deferred</td></tr>
<tr><td colspan="2">Requirement:
The system shall be able to fully program an ECU via CAN.

Rationale:
Part of the functionality of the program should provide the ability to follow a programming routine to program an ECU via CAN.

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td>Req: Sys:010</td><td>Hardware Support</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Requirement:
The system shall work with Peak CAN USB and Vector USB devices

Rationale:
The system must interface with existing CAN device hardware so that specific hardware does not have to be developed</td></tr>
</table>

Req: Sys:011b	Linux Support
Status:	Deferred

Requirement:
The system should work on Linux platforms

Deferral:
This is not included for V.0.1 of the project

Req: Sys:008	Open Source Licencing
Status:	Accepted

Requirement:
The system and any code shall be released under the MIT licence.

Rationale:
The code wants to be used by the wider community. By releasing it as open source it means that people can also contribute and improve the code.

Req: Sys:011	Platform Support
Status:	Accepted

Requirement:
The system shall work on windows platforms, specifically windows 7, 8, 8.1 and 10.

The system should support Linux platforms.

Rationale:
The company primarily uses Windows; however, it would be nice if the system were cross platform, but this is not a necessity.

Req: Sys:009	Software Language
Status:	Accepted

Requirement:
The system code shall be written in Python 3.

Rationale:
Python is a popular language. Specifying the language at the system level and attempting to use a single language minimises potential operability problems. As this is a monolithic tool specifying the language is important.

Req: Sys:001	Support for CAN transport protocol ISO 15765
Status:	Accepted

Requirement:
The system must support the transfer of data using the CAN Transport Protocol as defined in ISO 15765.

Rationale:
As CAN has a maximum payload size of 8 bytes, transferring larger numbers of bytes between the ECUs or tester to ECU requires some kind of higher-level transport protocol.
ISO 15765 defines a standardised CAN transport protocol which is supported by the majority of vehicle ECUs.

Req: Sys:014e	Support for Clear Diagnostic Information Service
Status:	Deferred

Requirement:
The system shall support the Clear Diagnostic Information Service

Rationale:
This is necessary for general maintenance on vehicle ECUs which use DTCs

Deferral:
This is not included for V.0.1 of the project

Req: Sys:003	Support for DoIP Transport
Status:	Deferred

Requirement:
The system shall support UDS communication over DoIP connections

Deferral:
This is not included for V.0.1 of the project

Req: Sys:002	Support for FlexRay Transport
Status:	Deferred

Requirement:
The system shall support UDS communication over the FlexRay connections

Deferral:
This is not included for V.0.1 of the project

Req: Sys:005	Support for K-Line Transport
Status:	Deferred

Requirement:
The system shall support communications over K-Line connections

Deferral:
This is not included for V.0.1 of the project

Req: Sys:004	Support for LIN Transport
Status:	Deferred

Requirement:
The system shall support communication over LIN connections

Deferral:
This is not included for V.0.1 of the project

Req: Sys:015a	Support for OBD Protocol Mode 1
Status:	Deferred

Requirement:
The system shall support the OBD-II mode 1 PID.

Rationale:
This is a standardised diagnostic format and is a common way to interface with vehicles and get useful information such as ignition timing or engine speed.

Deferral:
This is not included for V.0.1 of the project

Req: Sys:015b	Support for OBD Protocol Mode 2

<table>
<tr><td>Status:</td><td>Deferred</td></tr>
<tr><td colspan="2">Requirement:
The system shall support the OBD protocol mode 2 PID.

Rationale:
The PID mode 2 allows decoding of the freeze frame data which will allow a technician to figure out what caused a DTC code to be triggered.

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td>Req: Sys:015c</td><td>Support for OBD Protocol Mode 3</td></tr>
<tr><td>Status:</td><td>Deferred</td></tr>
<tr><td colspan="2">Requirement:
The system shall support the OBD protocol mode 3 PID.

Rationale:
The OBD Mode 3 PID allows a technician to read DTC codes.

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td>Req: Sys:015d</td><td>Support for OBD Protocol Mode 4</td></tr>
<tr><td>Status:</td><td>Deferred</td></tr>
<tr><td colspan="2">Requirement:
The system shall support the OBD protocol mode 4 PID.

Rationale:
The OBD Mode 4 PID allows a technician to clear DTCs.

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td>Req: Sys:015</td><td>Support for OBD Protocol PIDs</td></tr>
<tr><td>Status:</td><td>Deferred</td></tr>
<tr><td colspan="2">Requirement:
The system shall support OBD-II diagnostic modes

Rationale:
The OBD-II standard defines standardised diagnostic modes which are useful for diagnostics and information. These are especially useful for hobbyists who can use this information in other projects

Deferral:
This is not included for V.0.1 of the project</td></tr>
</table>

<table>
<tr><td>Req: Sys:014g</td><td>Support for Read Data By Identifier service</td></tr>
<tr><td>Status:</td><td>Accepted</td></tr>
<tr><td colspan="2">Requirement:
The system shall support the Read Data By Identifier service as defined in ISO-14229</td></tr>
</table>

Req: Sys:014f	Support for Read DTC Information Service
Status:	Deferred

Requirement:
The system shall support the Read DTC Information Service

Rationale:
This is necessary for general maintenance on vehicle ECUs which use DTCs

Deferral:
This is not included for V.0.1 of the project

Req: Sys:014b	Support for Routine Control Service
Status:	Deferred

Requirement:
The system shall support the Routine Control service defined in ISO 14229.

Deferral:
This is not included for V.0.1 of the project

Req: Sys:014a	Support for Security Access Service
Status:	Deferred

Requirement:
The system shall support the Security Access routine

Rationale:
This is necessary for programming an ECU or changing to certain defined ECU sessions

Deferral:
This is not included for V.0.1 of the project

Req: Sys:014d	Support for Session Control service
Status:	Deferred

Requirement:
The system shall support the Session Control service as defined in ISO-14229

Deferral:
This is not included for V.0.1 of the project

Req: Sys:014c	Support for Transfer Data service
Status:	Deferred

Requirement:
The system shall support the Transfer Data service as defined in ISO 14229

Deferral:
This is not included for V.0.1 of the project

Req: Sys:014	System compliance to ISO 14229
Status:	Accepted

Requirement:
The system shall comply to ISO-14229 - Unified Diagnostic Services

Rationale:
The system needs to be able to communicate with a wide variety of industry ECUs, therefore it has to comply to the defined industry standards otherwise interoperability problems may occur.

Req: Sys:007	Use of Open Source Code
Status:	Accepted
Requirement: The system should use Open Source software components where possible Rationale: As the software is open-source there should be little difficulty using other open-source projects and the more involvement in other open-source projects may result in further improvement for those packages	

Req: Sys:006	Use of OTS components
Status:	Accepted
Requirement: The system should use OTS components where possible Rationale: This is an attempt to lower cost and speed up development time. Where possible existing solutions should be considered to designing bespoke software	

Req: Sys:011a	Windows Support
Status:	Accepted
Requirement: The system shall support operation on the Windows platform, specifically Windows 7, 8, 8.1, and 10.	

Appendix E – System Behaviour Diagrams

These diagrams will add context and provide the reader with information on a domain and protocol to which they may be unfamiliar.

E.1. Uds Request & Response

Figure 14 shows a typical UDS request communication flow. When a request is transmitted from the tester, the sender the tester will either exit immediately (if no error has occurred in the transport) if the request does not require a response, or it will begin waiting for the response.

This starts an overall timeout which if exceeded will raise a timeout error, it will then proceed to check the response. If the response is a negative response it will retrieve the relevant negative response error and then raise it. After that it will decode the response. If the response is incorrectly formatted it will raise an incorrect formatting error.

If everything completes successfully it will return the encoded response from the ECU to the user.

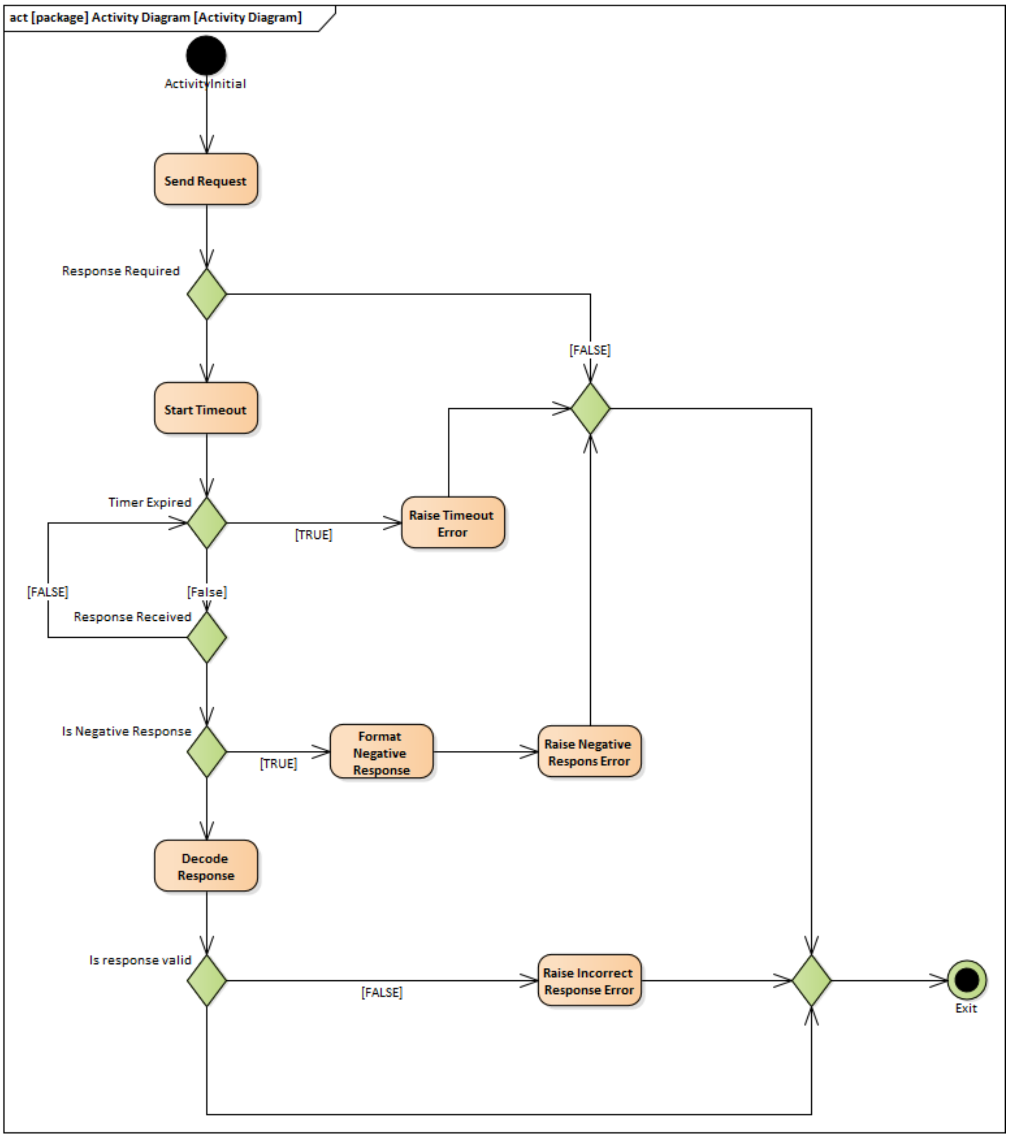


Figure 14 - A typical activity diagram for a UDS request

E.2. ECU Programming

Figure 15 is an example of a typical ECU programming sequence. Some of the steps have been removed from the standard E400 sequence as they are involved with checking serial numbers and part numbers to ensure that the correct secondary bootloader is programmed.

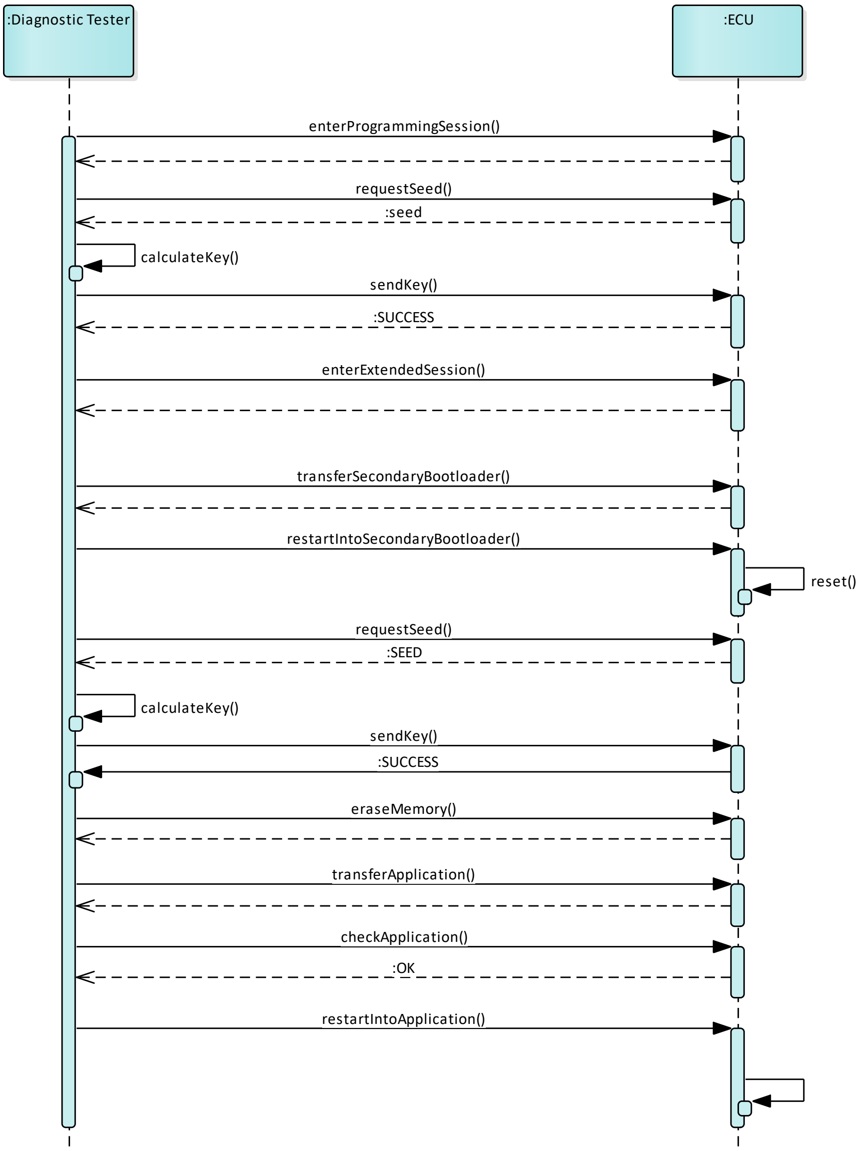


Figure 15 - ECU programming sequence

This sequence requires several interfaces, most of the communication is via the communications interface, but the calculation of the security key is done via an internal library.

In most cases these are either a DLL, a C library, or C source code. Python is able to interface with all of these via the ctypes interface.

Appendix F – System Architecture

This section provides an architecture overview of the system.

F.1. System Decomposition View

This view shows the system broken down into its component structure

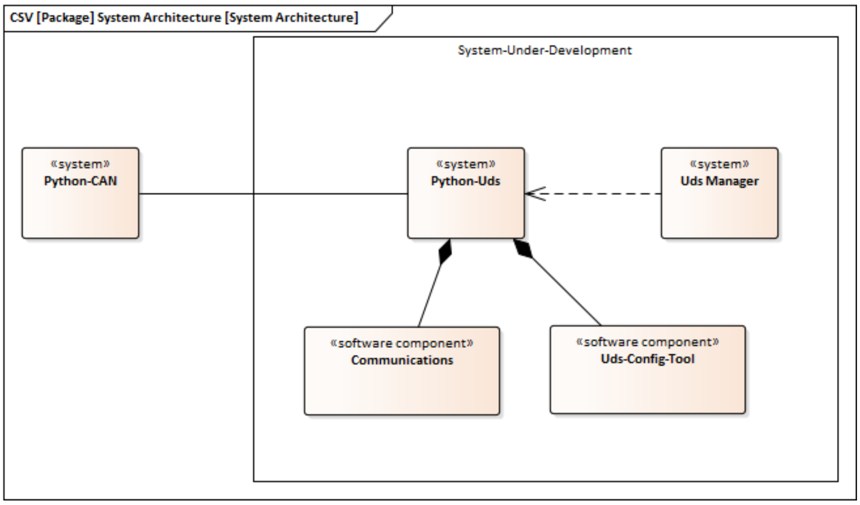


Figure 16 - The systems and their corresponding software components

The Python-Uds system comprises of two software components; Communications and Uds-Config-Tool. A separate system Uds Manager does not yet have any sub-components.

F.2. System Component View

This view shows the system broken down into its component structure, as the UDS manager does not yet have any components it is omitted from the following sections

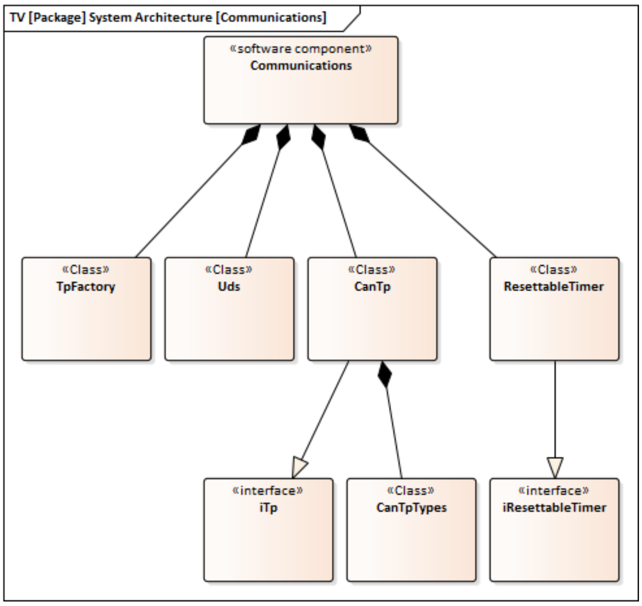


Figure 17 - The classes belonging to the Communications component and their composition

F.2.1. Communication

This section defines the structural and behavioural aspects of the classes in the UDS Communications component.

F.2.1.1. Uds

This section defines the structure and behaviour of the Uds class.

F.2.1.1.1. Structure

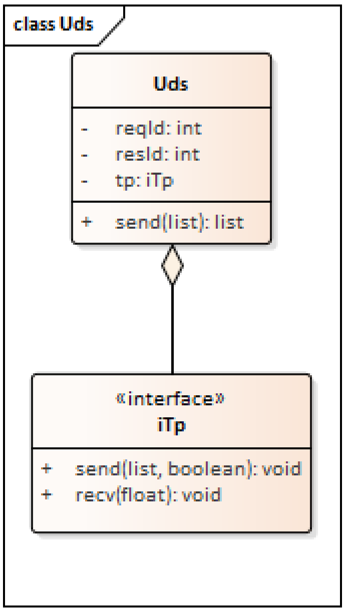


Figure 18 - Interface diagram for Uds

The UDS class has one public method, send() and has an attribute *tp* of type iTp.

F.2.1.1.2. Behaviour

F.2.1.1.2.1. Send

This is a simplified version of the actual UDS Request behaviour as the Uds send method simply sends a UDS PDU and (optionally) receives the response without performing any validation as the response is specific to the service. Each specific service as defined in the Uds-Config-Tool is responsible for these checks.

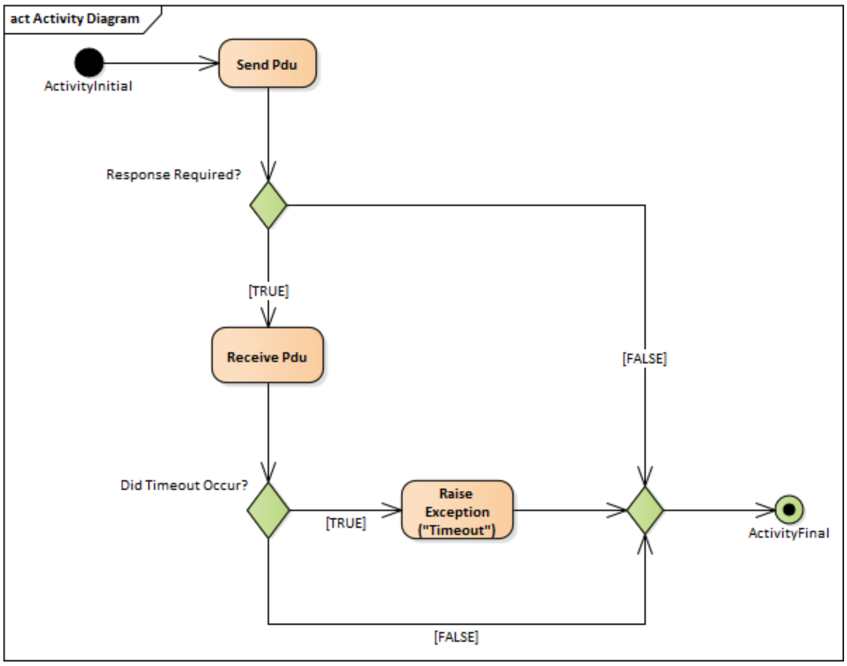


Figure 19 - Basic UDS request activity diagram

It can be seen as a wrapper for the iTp send and receive function, but performs nominally when used in the larger system context.

F.2.1.2. CanTp

This section defines the structure and behaviour of the CanTp class.

F.2.1.2.1. Structure

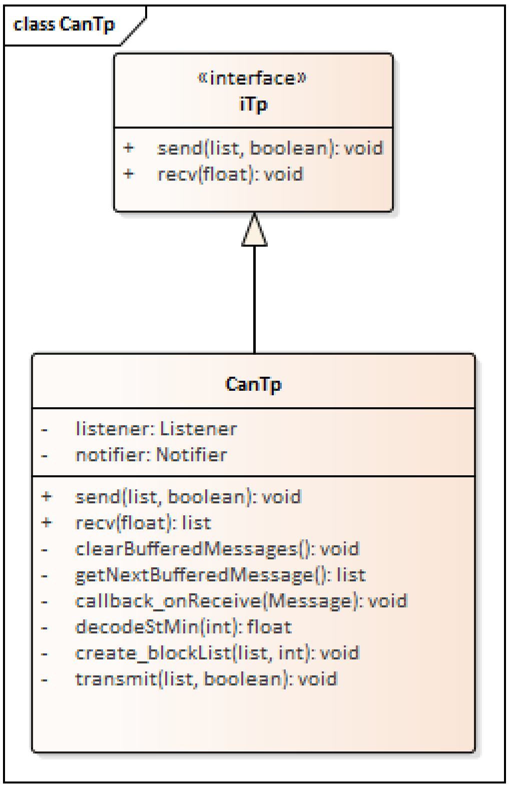

class CanTp
«interface»
iTp
+ send(list, boolean): void
+ recv(float): void
CanTp
- listener: Listener
- notifier: Notifier
+ send(list, boolean): void
+ recv(float): list
- clearBufferedMessages(): void
- getNextBufferedMessage(): list
- callback_onReceive(Message): void
- decodeStMin(int): float
- create_blockList(list, int): void
- transmit(list, boolean): void

Figure 20 - CanTp structural definition

The CanTp class has two public methods, *send()* and *recv()* all of the other methods are private. There are a set of enumerations which are specific to this class but they are not defined in this document, they are contained in the \uds_communications\CanTpTypes.py file in the repository.

F.2.1.2.2. State Machines

Two state machines are defined for CanTp; sending and receiving.

F.2.1.2.2.1. Sending

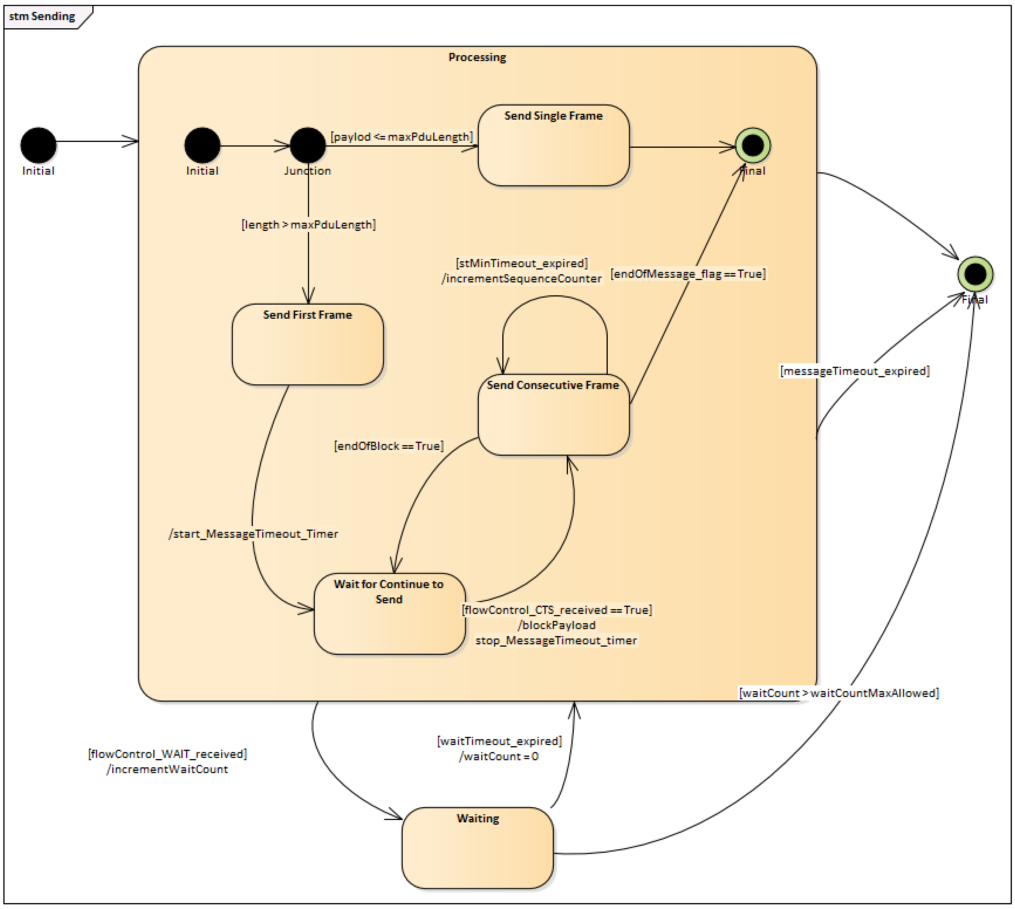


Figure 21 – Sending state machine for CanTp

The CAN Transport protocol has two major states, either sending or receiving. When in the sending state the system periodically listens for input from the server to control the message flow.

F.2.1.2.2.2. Receiving

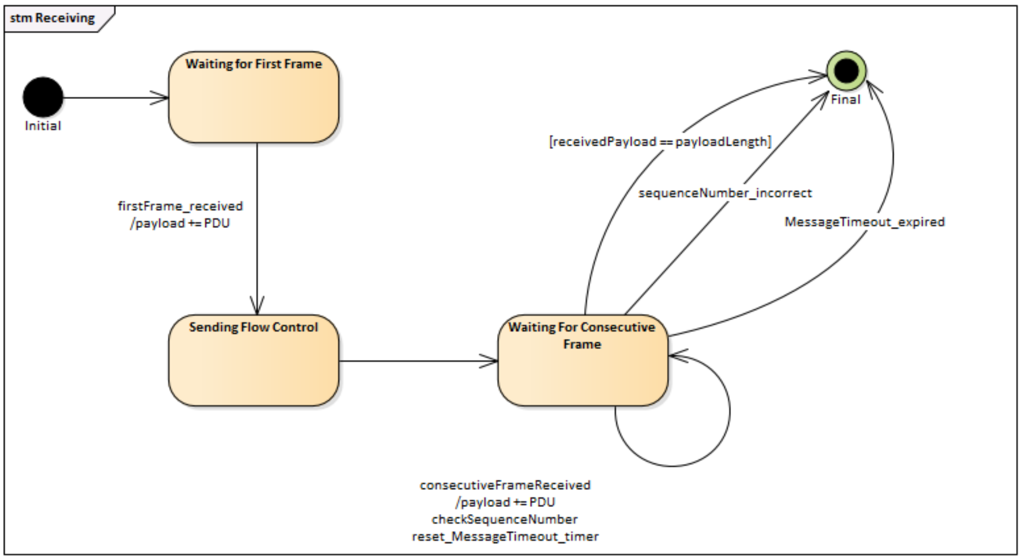


Figure 22 - State machine for the receiving of CAN TP messages

During the reception of a CAN TP message a large amount of the behaviour specified in ISO 15765 can be ignored as a PC has an almost infinite amount of memory in comparison to an automotive ECU, so none of the flow control behaviour is necessary and the block size can be set to maximum. This simplified the state machine in comparison to Sending as when the ECU is sending large payloads it must conform to the constraints of the target ECU.

F.2.1.3. Resettable Timer

The resettable timer is a simple class which provides a passive timer object which can be reset, start and stopped repeatedly.

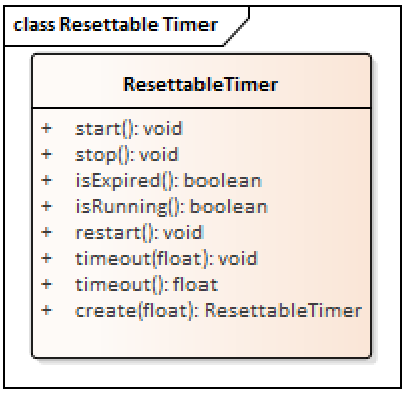


Figure 23 - Interface and class definition for ResettableTimer

During the construction of the CanTp class it became obvious that a resettable timer was needed. The behaviour needed to be non-blocking as messages could be received which would need to alter the state so that it could be stopped or restarted.

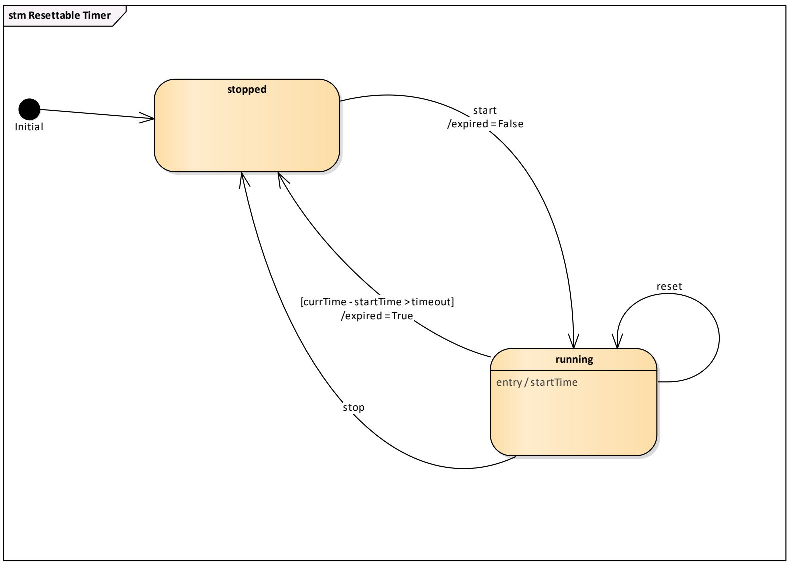


Figure 24 - State machine for the resettable timer

This timer will facilitate the requirements for the message timeout, wait and separation time timing needs. After analysing the problem, the idea that there might also be an expired state occurred, but it seemed to add an extra level of complexity so instead of an actual state an attribute was added to indicate that the timer ran to completion and was not prematurely stopped.

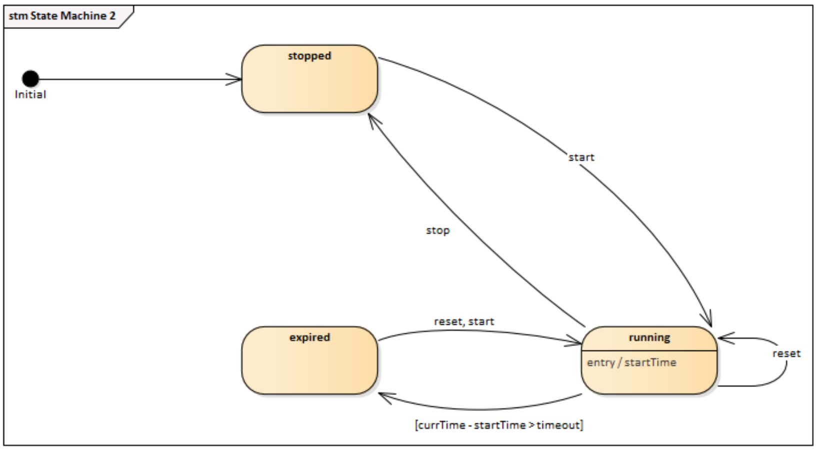


Figure 25 - Alternative state machine for resettable timer

The primary reason for not using this was that the start and reset triggers come transition from multiple states which seems overly complicated.

A further study of the timer experiments is available in Appendix H

F.3. Uds Config Tool

F.3.1. Structure

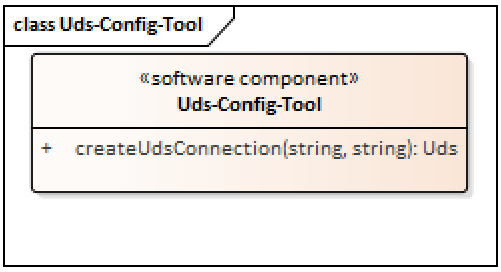


Figure 26 - Initial interface and structure for the Uds-Config-Tool

A definite interface for the Uds-Config-Tool has not yet been finalised, however certain structural decisions can be made with the information about the domain. The class will either be a singleton, or potentially a collection of standalone methods. Its primary method at present is the *createUdsConnection* method in \uds_config_tool\UdsConfigTool.py. This method iterates over the XML file extracting the DIAG_SERVICE and uses the relevant Function Creation and Supported Service classes to dynamically generate the necessary functions.

It returns a Uds object which has all of the relevant containers and methods bound to the instance.

F.3.2. Generic Service Behaviour

All of the UDS services have a similar structure and communicate in a very similar way. A generic template was created to define the stages of a UDS request.

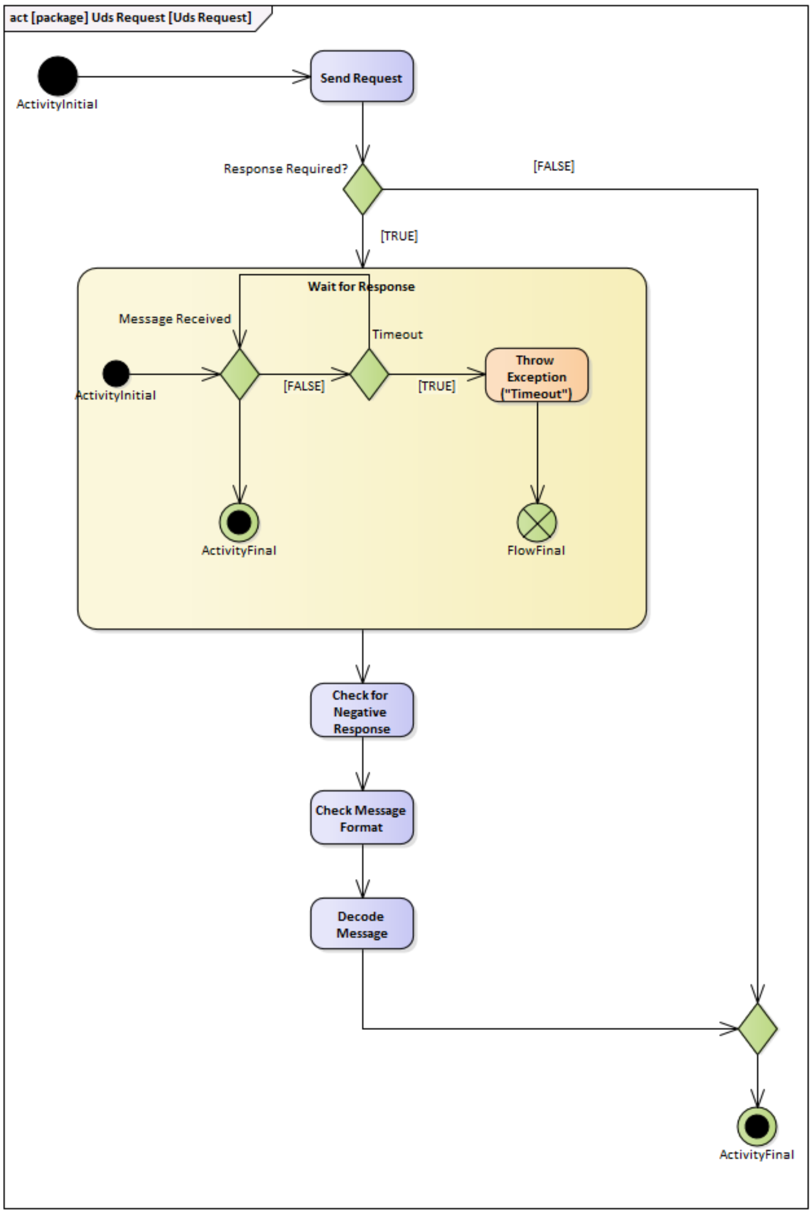


Figure 27 - Generic UDS request when using the Uds-Config-Tool code

The actions in purple are the dynamically generated functions as their content changes based on the service. For example, the Read Data By Identifier (RDBI) service has a response service ID of 0x62, while the Write Data By Identifier (WDBI) service response service ID is 0x6E, so the Check Message Format action has to change depending on the service, and each of the parameters for each service will have different constraints for length, etc. Using that same example, the request RDBI takes no input parameters, but WDBI does, so the request function would be different for the two services as it would be necessary to encode the input for the WDBI service and check it is valid.

This template works for each of the UDS services as they all have these steps but the procedure for each step is not generic. This is expanded on further in the following sub-section.

F.3.3. Read Data By Identifier

This section describes the behaviour of the Read Data By Identifier Service and how this works with the Uds-Config-Tool.

F.3.3.1. Behaviour

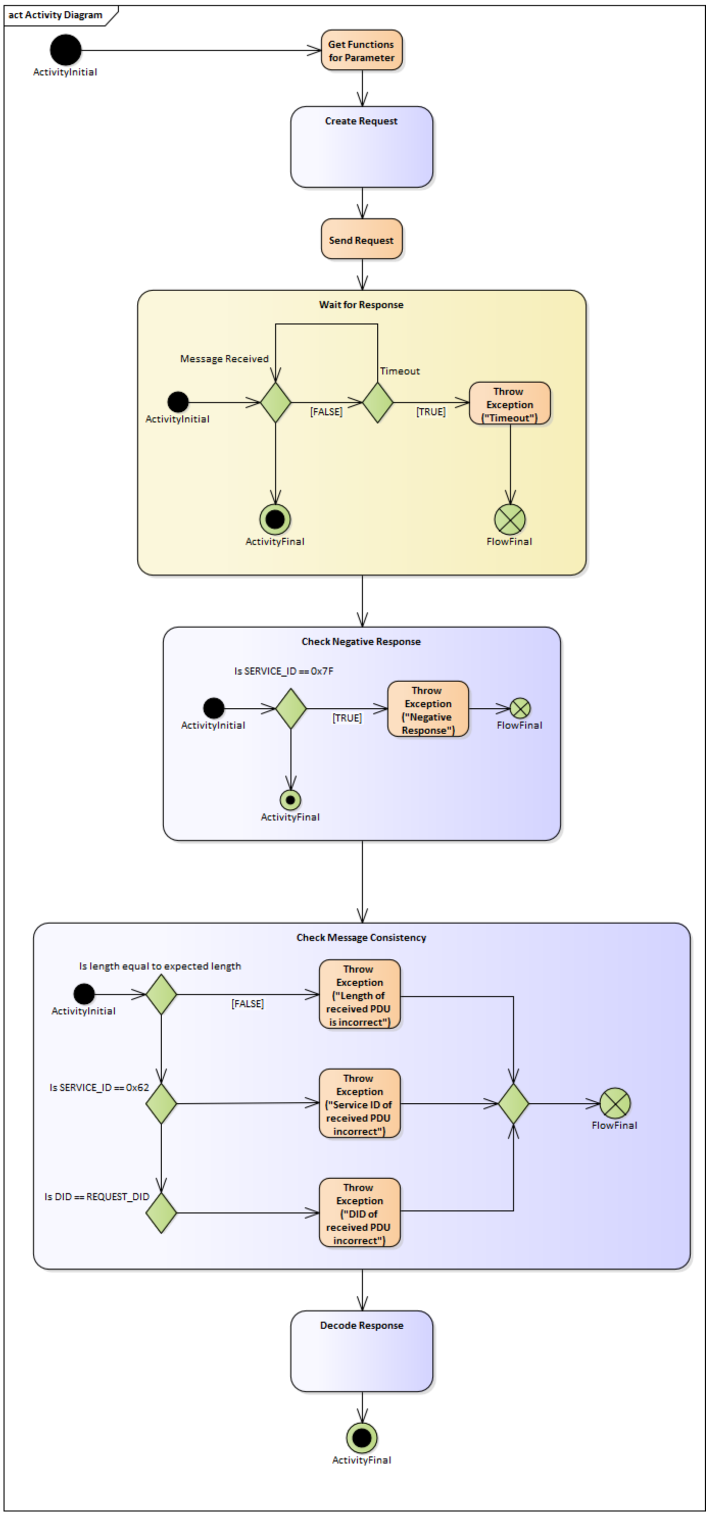


Figure 28 - Behaviour of the ReadDataByIdentifier function

This diagram is not intended to be syntactically correct to UML standards, but more as a graphical pseudocode representation of what is required of the Read Data By Identifier method. The Activities in purple are the dynamic functions. When the method is called the functions are retrieved from their respective containers and stored for execution.

The create request function formats the request which in the case of the ReadDataByIdentifier service takes no inputs and simply creates the request payload. For example [0x22, 0xF1, 0x8C] is the PDU which represents the request for an Ecu Serial Number.

The message is transmitted using the send method of the Uds class, and in this case a response is always expected so the next activity is waiting for the response. If a response is not received in the timeout period an exception is raised.

When a PDU is received the next stage is to check that the payload is a negative response. If the payload is a negative response then an exception is raised.

Next is a check of the consistency of the message to make sure that it is returned as expected. The length of the PDU, the response ID and response DID are checked and if any of these are not as expected a relevant exception is raised.

Finally the PDU is decoded from its raw format into the defined format. In the case of ECU Serial Number for the E400 bootloader the response is a 16 byte ASCII string.

Appendix G – Functional Programming Research

An attempt was made during this project to experiment with functional programming techniques to solve some of the problems. Functional programming is known to produce very succinct code however some people find it hard to read and understand as it relies on more complicated constructs such as recursion. A few simple tests were carried out to experiment with the benefits and disadvantages of functional programming.

It is worth noting that Python is not a functional programming language, and lacks several important features such as infinite recursion depth. It will throw errors if a function attempts to recurse past a certain defined limit. This limit can be extended, but as it is not always known to what depth a particular input to a program will require, this means that the behaviour can be non-deterministic.

One requirement of the UdsConfigTool is to be able to decode the payloads returned by the server, and one typical example would be a multi byte integer. For example, running time could be represented by a 32-bit integer, this would be transmitted as 4 individual bytes.

```
     [0x00, 0x00, 0x43, 0x32]
```

This would equal 0x4332 or 17202 seconds. A simple function is required to take the 4-byte payload and represent it as its 32-bit integer.

G.1. Methods

Three functions were created to test this, a non-recursive function, an explicitly recursive function and a function implementing the *reduce* method from the python functools library.

Non-recursive Function

```
    def buildIntFromList(aList):
        result = 0
        for i in range(0, len(aList)):
            result += (aList[i] << (8 * (len(aList) - (i+1))))
        return result
    return buildIntFromList(aList)
```

This is using a for loop to iterate over the list, appropriately shift the value into its correct position and returns the result.

Explicitly Recursive Function

```
    def buildIntFromList(aList):
        if(len(aList) == 1):
            return aList[0]
        else:
            return (aList[0] << (8 * (len(aList) - 1) )) + 
buildIntFromList(aList[1:])
    return buildIntFromList(aList)
```

This function is a recursive function where the if statement checks for the base case and the else contains the recursive case.

Reduce Method

```
    def buildIntFromList(aList):
        return reduce(lambda x, y: (x << 8) + y, aList)
```

This function utilises the *reduce* function from the python functools library. This is a higher-order function used to manipulate variables as a single unit.

G.2. Results

The cProfile library was used to test these three functions with a test array of 900 elements. In the case of the recursive function it had to be declared inside of another function as it would otherwise run 500 separate profile instances. It was decided that all of the code would be arranged similarly to make the results as consistent as possible

For Loop

2504 function calls in 0.001 seconds
Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0	0	0.003	0.003	main.py:25(nonRecursiveFunc)
1	0.003	0.003	0.003	0.003	main.py:27(buildIntFromList)
2501	0	0	0	0	{built-in method builtins.len}
1	0	0	0	0	{method 'disable' of '_lsprof.Profiler' objects}

Recursive Function

7501 function calls (5002 primitive calls) in 0.047 seconds
Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0	0	0.047	0.047	main.py:16(recursiveFunc)
2500/1	0.047	0	0.047	0.047	main.py:18(buildIntFromList)
4999	0	0	0	0	{built-in method builtins.len}
1	0	0	0	0	{method 'disable' of '_lsprof.Profiler' objects}

Reduce Function

2503 function calls in 0.003 seconds
Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0	0	0.003	0.003	main.py:34(reduceFunc)
1	0	0	0.003	0.003	main.py:36(buildIntFromList)
2499	0.002	0	0.002	0	main.py:37(<lambda>)
1	0	0	0.003	0.003	{built-in method _functools.reduce}
1	0	0	0	0	{method 'disable' of '_lsprof.Profiler' objects}

The recursive function is almost an order of magnitude slower than the for loop and the reduce function. This was later concluded to be due to the function call overhead. It also has to execute the *len()* function many more times as every time the loop executes two calls are made (except in the final instance), but the results imply that the cumulative time to execute the *len()* function is negligible.

All three of these functions have the same time complexity, O(n), but the performance is different. As these functions are not intended to be executed on a time-sensitive schedule, and in practice the largest input would likely be 8 bytes (most integers used in automotive design tend to stick to 32 bit or 64-bit maximum due to the processing overhead dealing with larger numbers and the lack of need for anything larger) so the difference in performance is negligible, therefore the primary concern is of readability and maintainability.

Another example of this would be the conversion of an integer list to a String. As most of the serial numbers used in automotive ECUs are ASCII coded it is likely that a 16 to 32-digit serial number would be returned from the ecuSerialNumber parameter.

```
[0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30, 0x30, 0x35, 0x39]
```

Would equal:

```
"0000000000000059"
```

To convert the byte list to a string would be similar to creating the integers, any one of the three methods could be employed. Tests were performed on methods to convert these lists and the results were very similar with the recursive function taking 50 times longer to execute. From this it was deduces that the main time taken in the recursive case is the function call overhead. It is also worth noting that to get enough resolution for the string tests the size of the array was increased to 2500, and the recursion depth had to be increased.

In conclusion, it was an interesting learning experiment, and the use of the reduce method and the lambda functions were very succinct, but in these instances not much is gained over using a simple for loop. The for loop is significantly easier to understand than either the recursive or reduce methods and does not require any external libraries (unlike the reduce method which relies on the functools library).

As one of the project objectives is education the reduce method will be implemented for the project, and this will be commented to allow others to learn from this experience.

The code for these tests can be found in the test/profiling folder of the uds-config-tool repository.

Appendix H – Threading and Multiprocessing Timer Tests

Three different timer classes were created to attempt to solve the problem of non-blocking pauses in the code execution. A 'manual' timer where a function must be called to see if the timer has expired, a 'threaded' timer where the timer will automatically expire after a set period, and a 'multiprocess' timer where a separate process is spawned to monitor the timer for expiry.

The code can be seen in the \experiments\timers\ folder in the code repository.

4 Experiments were conducted to see which would be the most effective method. The criteria measured are: accuracy and precision (ISO, 1994). Each of the 4 classes inherited a standardised interface iTimer which defined their interface methods, primarily the only one of interest is the *isExpired* method which returns True if the timer has run to completion.

The 4 experiments were:

1. ResettableTimer – This class has a method *isExpired*() which when called checks the current time against the start time and if the difference exceeds the timeout period then it returns True, otherwise it returns false. This is the simplest implementation but was thought to possibly cause problems as the code has to be executed in the main task which may add extra processing overhead.
2. ThreadTimer – This class utilises the *threading.Timer* class to set up a separate thread which when completed sets the *__expired_flag* variable to True, therefore when the *isExpired()* method is called it simply returns the current state of the variable. This is more complex than the ResettableTimer as it uses threading. The idea is that by spawning a separate thread this might allow the underlying interpreter to process this in parallel with the main task. For each start of the timer a new instance of the Timer class has to be created which might increase the amount of "garbage" generated which needs to be cleaned up by the underlying interpreter.
3. ManualThreadTimer – This class sets up a separate monitoring thread which sits in `while(1)` loop. This continuously monitors the timer and sets the *__expired_flag* variable to True when the timer is expired, therefore the *isExpired()* method simply returns the state of the variable. The idea is that by spawning a separate thread this might allow the underlying interpreter to process this in parallel with the main task. The difference to the ThreadTimer is that it is slightly cleaner than using the threading.Timer class and might reduce the number of created instances.
4. MultiprocessingTimer – This class sets up a completely separate process which inside it contains a thread which works similarly to ManualThreadTimer. The idea was that the separate process may be able to be executed by a separate processor core and spread the processing load over multiple cores.

Each of the timers was tested using the following test code:

```
a = timer(0.001)

results = []
for i in range(0, 100):
    startTime = perf_counter()
    a.start()
    while (a.isExpired() == False):
        pass
    endTime = perf_counter()
    delta = endTime - startTime
    results.append(delta)

print("Min: {0}".format(min(results)))
print("Max: {0}".format(max(results)))
print("Avg: {0}".format(sum(results)/len(results)))
```

The results from each of the tests are as follows:

Type	Min	Max	Average
ResettableTimer	0.001	0.001	0.001
TheadTimer	0.0095	0.0171	0.0155
ManualThreadTimer	0.001	0.0011	0.0011
MultiprocessingTimer[7]	0.001	0.002294	0.001314

Table 3 - Timing figures for 100 samples of the different timer implementations

All of the implementations except the ThreadTimer use the time.perf_counter method which gives very high resolution timing.

From the results it is obvious that the ThreadTimer is not acceptable as it is neither accurate or precise. It is possible that this can be attributed to the use of the time.time method which as noted in the report has a limited resolution of ±16ms due to limitations in the Microsoft Windows timer functionality.

The other three are comparable with the MultiprocessingTimer being the least accurate of the three. The multiprocessing timer did have other problems in that it was more difficult to communicate the state of the variables between the processes and a more sophisticated mechanism might be required. The ResettableTimer and ManualThreadTimer perform better in terms of accuracy and precision, and are the simpler implementations.

These tests were conducted with 10000 samples, but the results were not very different, with the ResettableTimer and ManualThreadTimer being the most consistent.

Type	Min	Max	Average
ResettableTimer	0.001	0.001	0.001
TheadTimer	0.005	0.085	0.0158
ManualThreadTimer	0.001	0.004	0.0011
MultiprocessingTimer	0.001	0.0114	0.0024

Table 4 - Timing figures for 10000 samples of the different timer implementations

It was decided that the ResettableTimer would be used for this project as it was the simplest, most accurate and most precise.

[7] The results for this had to be obtained from data reported by a print statement inside the thread. The implementation did not allow the data to be collected by the standard test code.

Appendix I – Dynamic Function Generation Example

The example chosen for this is the parameter for 'Boot Software Identification' as it has two return parameters and most others simply have one. For this example, we will reference the "Bootloader.odx" file included with this report. This is also available in the repository in the \experiments folder. We will be referencing the ReadDataByIdentifierMethodFactory class located in \uds_config_tool\FunctionCreation\ReadDataByIdentifierFactory.py

The definition for the parameter is located on line 5266. From this we can get the name of the parameter, and the references for the Request, Positive Response and Negative Response.

As defined in the activity diagram in Appendix F.3.3 the steps include; Request, Negative Response Check, Consistency Check, Decode Positive Response.

I.1. Request

For the Request step we go to the Request entry at line 7305. This gives us the name for the function, the Service Id to use, and the Parameter that will be Transmitted. This request will encode to a [0x22, 0xF1, 0x80]. To do this we use the create_requestFunction() method.

For each parameter in the PARAMS tag, it iterates over the entries and determines how to decode them. For the SERVICE-ID semantic it simply converts the value in CODED-VALUE into an int. For the ID semantic it uses the intArrayToIntArray() method to turn an array of a single 16 bit number and splits it into its 8 bit parts, one high bit and one low bit.

These two values are stored as separate parameters, and then substituted into the requestFuncTemplate

```
def {0}():
      return {1} + {2}
```

This produces the following code when the string is completed.

```
def request_RQ_Boot_Software_Identification_Read():
      return [0x22, 0xF1, 0x80]
```

I.2. Negative Response Check

For the negative response we go to the entry at line 11523. This gives us the name for the function, the SERVICE-ID which in this case is 0x7F, the Service ID of the requesting service and the returned negative response code. The purpose of this function is to check for the presence of the negative response and raise an exception if this is detected. To do this we use the create_checkNegativeResponseFunction() method.

The negative response decoding is primarily interested in the param with the SERVICE-ID semantic and its position. It gets this information add adds it to the checkString template.

```
    if input[{0}:{1}] == {2}: raise Exception(\"Detected negative response: [hex(n) for n in input]\")
```

This then becomes

```
    if input[0:1] == 127: raise Exception(\"Detected negative response: [hex(n) for n in input]\")
```

This string gets substituted into the negativeResponseFuncTemplate

```
    def {0}(input):\n
        {1}
```

Which becomes

```
    def check_negResponse_NR_Boot_Software_Identification_Read(input):\n
        if input[0:1] == 127: raise Exception(\"Detected negative response: [hex(n) for n in input]\")
```

As this functions purpose is to raise an exception if a negative response is detected, it does not need to return anything. The exception raised includes the entire payload of the response to give the user more information for debugging

I.3. Consistency Check

For the consistency check we go to the entry at line 9415. This gives us the name for the function, and the format of the returned parameter which includes all of the sub-parameters. To do this we use the create_checkPositiveResponseFunction() method.

The consistency check is mainly interested in checking; the returned service id is correct, the DID is the expected value, and the length of the payload is as expected. If any of these are incorrect then the system raises a relevant exception. It iterates over the entries in the PARAMS tag and retrieves the relevant information.

- For the service ID it extracts the value at position [0:1]
- For the DID it extracts the bytes at position [1:3]
- For each DATA entry it computes the length for the entry in that section, however to do this it must traverse into the DATA-OBJECT-PROPS entry in the DIAG-DATA-DICTIONARY-SPEC entry to get the relevant DATA-OBJECT-PROP. For this example it has to retrieve the entries at line 232 and line 2950. These provide the information on how to decode a 1 byte hex value, and a 24 byte ASCII string respectively.

Each of these values is substituted into the checkFunctionString template

```
def {0}(input):\n
      serviceIdExpected = {1}
      diagnosticIdExpected = {2}
      serviceId = DecodeFunctions.buildIntFromList(input[{3}:{4}])
      diagnosticId = DecodeFunctions.buildIntFromList(input[{5}:{6}])
      if(len(input) != {7}): raise Exception(\"Total length returned not as expected. Expected: {7}; Got
      {{0}}\".format(len(input)))
      if(serviceId != serviceIdExpected): raise Exception(\"Service Id Received not expected. Expected {{0}};
      Got {{1}} \".format(serviceIdExpected, serviceId))
      if(diagnosticId != diagnosticIdExpected): raise Exception(\"Diagnostic Id Received not as expected.
      Expected: {{0}}; Got {{1}}\".format(diagnosticIdExpected, diagnosticId))
```

This would become

```
def check_PR_Boot_Software_Identification_Read(input):
    serviceIdExpected = 98
    diagnosticIdExpected = 61824
    serviceId = DecodeFunctions.buildIntFromList(input[0:1])
    diagnosticId = DecodeFunctions.buildIntFromList(input[1:2])
    if(len(input) != 29): raise Exception(\"Total length returned not as expected. Expected: 29; Got
    {{0}}\".format(len(input)))
    if(serviceId != serviceIdExpected): raise Exception(\"Service Id Received not expected. Expected {{0}};
    Got {{1}} \".format(serviceIdExpected, serviceId))
    if(diagnosticId != diagnosticIdExpected): raise Exception(\"Diagnostic Id Received not as expected.
    Expected: {{0}}; Got {{1}}\".format(diagnosticIdExpected, diagnosticId))")
```

This is the most complicated check. It could be argued that this could be performed with the decode step, but it was decided that each operation would be atomic even if the code is duplicated slightly.

I.4. Decode Positive Response

For the decoding of the positive response we use the entry at line 9415 again as with the consistency check. This gives us the name for the function from the SHORT_NAME tag, and the format of the parameters that have been returned. This solely interested in the data parameters. For each entry in the PARAMS tag with semantic DATA, its corresponding DIAG-DATA-DICTIONARY-SPEC entry is found. The encoding type is used to determine which functions are used to decode the values, currently decoding ASCII strings and integer data is supported. If the entry is an ASCII string, the intListToString function is used, otherwise the data is simply returned as a list.

The functionString template is used to create the decode method

```
DecodeFunctions.intListToString(input[{0}:{1}], None)
```

And

```
input[{1}:{2}]
```

For this parameter becomes:

```
DecodeFunctions.intListToString(input[4:29], None)
```

And

```
input[3:4]
```

These are added to a dictionary list *result* where the key is the Long Name for the relevant PARAM entry (lines 9437 and 9443) giving:

```
        result[‘numberOfModules’] = input[3:4]
```

and

```
        result[‘Boot Software Identification’] = DecodeFunctions.intListToString(input[4:29], None)
```

Finally these strings are added to the encodeFunctionString template

```
        def {0}(input):

                result = {{}}

                {1}

                return result
```

Giving:

```
        def encode_PR_Boot_Software_Identification_Read(input):
                result = {}
                result[‘numberOfModules’] = input[3:4]
                result[‘Boot Software Identification’] = DecodeFunctions.intListToString(input[4:29], None)
                return result
```

Each of these functions is added to a separate dictionary in the *ReadDataByIdentifier* container with a common key entry in each of the dictionaries so that they can easily be retrieved from the *readDataByIdentifier* method, so that when the user calls:

```
        Bootloader.ReadDataByIdentifier(‘Boot Software Identification’)
```

They get

```
        {'numberOfModules': 1, 'Boot Software Identification': '000000000000000000000001'}
```