POLITECNICO DI TORINO

Master Degree
in Mechatronic Engineering

Master Thesis

asn2c: Modern ASN.1 Compiler for Advanced Network Messages and Data Structures

Relatori
prof. Marco Rapelli
prof. Francesco Raviglione
prof. Claudio Ettore Casetti

Candidato
Francesco Di Gregorio

Anno Accademico 2024-2025

Acknowledgements

Prima di iniziare mi sembra giusto dedicare qualche frase a chi durante questo percorso mi è stato vicino e mi ha supportato.
Il primo ringraziamento va sicuramente ai professori Marco Rapelli, Francesco Raviglione e Claudio Casetti per avermi sempre dimostrato fiducia nel corso di questo progetto lasciandomi piena libertà su come portare avanti il lavoro ma dandomi consigli essenziali su come portarlo a compimento.
Un ringraziamento va a tutto il team Reply che mi ha sostenuto e aiutato durante i miei anni di studi qua a Torino, è anche grazie a voi se ho potuto portare a termine questo percorso in parallelo al lavoro.
Un ultimo ringraziamento, non per importanza, va a Matteo e Federico, i miei due coinquilini durante questi anni, siete stati dei perfetti compagni di viaggio. Una menzione d'onore va però a Matteo, l'idea per il nome del progetto è stata tua e di conseguenza questo progetto è anche un po' tuo.

Abstract

The automotive field is experiencing a strong technological evolution in recent years. This evolution is leading to the standardization of novel technologies for connected vehicles (both cellular and Wi-Fi-based), enabling them to exchange data with other vehicles and with the infrastructure, to make transportation safer, greener and more autonomous. All these communication technologies define standardized messages for different purposes, that include exchanging data between road users, but also transmitting management information and requesting certificates for secured data transfers. These messages and their content are usually defined as a set of data structures, in turn defined by the Abstract Syntax Notation One (ASN.1) description language. Starting from an ASN.1 definition, several tools enable the automatic generation of code for the encoding and decoding of the messages. One of the most used open-source tool available is asn1c, maintained by a vast community behind this project. In 2020 the European Telecommunications Standards Institute (ETSI) released new ASN.1 files describing the IEEE 1609.2.1. data structures used for certificate needed in the security application defined by IEEE 1609.2. At the moment of writing, the only existing software that are able to compile the ASN.1 files are licensed causing an important bottleneck for research. The objective of this thesis work is to develop an ASN.1 compiler that supports all the ETSI's specification. The developing processes started from the source code of asn1c. asn1c does not support the files related to security and certificate management due to the presence of complex structures adopted in the new release. This thesis propose a solution to overcome the described problem, developing a free and open-source tool able to manage all the ETSI's files, improving also the existing code generation layer present in asn1c.

Contents

Chapter 1

Introduction

In recent decades, the constant growth of vehicle ownership [1], has placed an increasing stress on road infrastructure and more complex challenges arise. [2]. To respond to such challenges, conventional methods cannot be taken into account, and more recent and advanced applications need to be implemented. In 2010, the European parliament adopted the directive 2010/40/EU, which "establishes a framework in support of the coordinated and coherent deployment and use of Intelligent Transport Systems within the Union" [2].
Directly from the EU directive, "Intelligent Transport System (ITS) means systems in which information and communication technologies are applied in the field of road transport, including infrastructure, vehicles and users, and in traffic management and mobility management, as well as for interfaces with other modes of transport" [2]. The European Commission with this document recognized that to solve the increasing traffic volumes and the correlated problems, such as a greater road congestion, energy consumption and environmental and social problems, innovations like ITS are needed. For instance, the road congestion economical impact to EU is estimated at €110 billion [3], and tragically every year more than 20000 lives are taken in road accident [4].
One of the foundation on which ITS is based is the concept of Vehicular-to-Everything (V2X) communications. V2X refers to wireless data exchange between a vehicle and everything in its environment that may influence, or be influenced, the vehicle's behavior. This includes Vehicle-to-Vehicle (V2V) communication, in which autonomously two or more vehicles exchange data between each other, Vehicle-to-Infrastructure (V2I) communication, in which the vehicle communicates with the road infrastructures (traffic lights, road signals) to gather data about road state, Vehicle-to-Person (V2P) communication, in which the data are exchanged between vehicles and the so called, vulnerable road user, like pedestrian and cyclist.
V2X communications can improve situational awareness and can help reach a higher level of autonomous driving by improving coordination between vehicles and allowing better exchange of critical information between all the involved agents.
In Europe, the European Telecommunications Standard Institute (ETSI) is leading the development and standardization procedure of V2X communication through its Technical

Committee on ITS (ETSI TC ITS) [5]. One of the principal projects in which ETSI is involved is the definition and of ETSI ITS-G5 standard. This new standard is the European profile for short-range vehicular communications based on IEEE 802.11p and adapted for ITS [6]. It operates in the 5.9 GHz frequency band specifically allocated in Europe for road safety and traffic applications. This technology is designed to support the exchange of standardized messages described by ETSI such as Cooperative Awareness Message (CAM) and Decentralized Environmental Notification Message (DENM) [7], [8]. Many additional messages have been standardized by ETSI, defying information exchanges among different road users and providing the foundation for security protocols that ensure safe vehicular communications.
ETSI employs Abstract Syntax Notation One (ASN.1) as the formal language to specify the structure and content of its messages. The use of this approach, allow to describe complex data structures, such as nested declaration, with precision. Moreover, the ASN.1 definitions are independent from both hardware and programming language, granting developers full freedom in selecting the technologies most suited to their needs while ensuring compliance with ETSI standard.

1.1 Research Motivation and Objectives

While ETSI's choice of using ASN.1 files to describe their messages format allow the developers to freely choose the platform on which build their ITS-G5 compliant applications, it also requires the use of an additional software responsible of the translation from ASN.1 into the target programming language. These software tools are commonly called ASN.1 compiler and they are in charge of the generation of the source code.
Given the rapid pace at which technologies evolve ,ETSI must continuously update its specifications. As result, every few years entirely new standard or revised version of existing ones are released, accompanied by ASN.1 files that need to follow the increasing complexity of each new release. In addition to this factor, every few years the International Telecommunication Union (ITU), who standardized ASN.1 format and it is in charge of its maintenance, release a new version of ASN.1, with the latest being described by ITU-T X.680 released in 2021 [9]. In order to handle ASN.1 files whose complexity increase over the years, the ASN.1 compilers need to updated so that they are able to understand the new structures and consequentially generate the source code able to fully represent the new requirements.
For commercial ASN.1 compilers this task is relatively straightforward, as they are maintained by a dedicated team that ensure compliance with the latest standard, the open-source compilers often rely on community contributions to remain up to date, which may results in features being not partially implemented or not implemented at all. This limitation constitutes the main motivation of the present thesis. The open source software *asn1c* compiler, available on GitHub, is largely used to generate C code from ASN.1 files. However, it still lack full support for several newly released files published by ETSI.
The object of this thesis is to analyze the current limitations of *asn1c* and develop a new ASN.1 compiler that will correctly handle all the ASN.1 files included in the official ETSI

repository.

1.2 Main contributions

The main object of this thesis is to analyze the *asn1c* compiler and its working principle to have a better understanding on how the code generation process is implemented and its limitations and, in the end, developing our own ASN.1 compiler. This first step is essential to reach the core of this thesis, understanding the limitation of the *asn1c* project is the foundation for identifying where the problems arises when compiling the ETSI ASN.1 files and for determining what is required to successfully generate correct C code. Furthermore, at the end of the thesis work various test are going to be performed to check the performance of the modified *asn1c* software.
More details about the topics covered in this thesis are listed below.

1.2.1 ASN.1 Specification and Current *asn1c* Status

As stated above the first step of this work consist in the study of the ASN.1 specification and how *asn1c* works. The main source of information on ASN.1 specification comes directly from the latest ITU recommendation ITU-T X.680 [9]. For all the information about *asn1c*, an official documentation is available on the project's GitHub page but only describe how to use the tool not how the main functions are implemented. For such reason the first contribute of this thesis is to analyze the source code of the ASN.1 compiler tool to have a better knowledge of it. The second part of the study on *asn1c* consist of hands on testing of the tool by generating code from both toy example and official ASN.1 files. This procedure aims to better understand its limitations and to identify which component of the code requires modification. Some of the issue analyzed in this thesis are already documented in the *Issue* section of the project's GitHub repository.

1.2.2 *asn1c* Implementations

Once the limitations of the *asn1c* software are documented, the main objective of this thesis is to implement the right modifications to fix these problems allowing the tool to work correctly despite the ASN.1 file given as input. Moreover, this section can be divided in two part separated by the logic applied to the corrections.
The first group consists of fixes applied directly to the source code. All the implementations included in this category are direct modification of the existing code. The majority of these changes consist of completing functions that are already partially implemented.
The other modifications to the *asn1c* project consist of adding new scripts designed to run before the compiler. This approach was selected to face certain issue in a simple manner, as the components involved represents core elements of the *asn1c* tool. To avoid introducing changes in such delicate parts, an higher level solution has been chosen.

1.3 Outline of the Thesis

The present thesis is divided into sections that describe, in chronological order, the entire course of the project. In particular:

- **Chapter** 2 introduces the ASN.1 standard and describe how a ASN.1 files it is built. In this chapter are also presented the main tool used to generate C code starting from ASN.1 files.
- **Chapter** 3 presents the main problems related to the project taken in analysis and which are the causes related to them.
- **Chapter** 4 describes the solution implemented to solve the issue that are presented in the previous chapter.
- **Chapter** 5 summarizes the activity done in this project, including the feature implemented and what can be implemented to further improve. In this chapter are also presented some suggestions for future works.

Chapter 2

ASN.1 Standard and ASN.1 Compilers

The starting project examined is the ASN.1 compiler "asn1c" from the github of mouse07410 [10]. The main purpose of this software is to parse files provided by ETSI, written using the Abstract Syntax Notation One (ASN.1) format, and to generate the corresponding .c and .h source files representing the defined data structures.

2.1 ASN.1 Format

The ASN.1 format was first standardized in ITU X.208 on 1988 [11],that recommendation is now deprecated and the official documents from International Telecommunication Union (ITU) that describe the ASN.1 standard are X.680-683, first published on 1995 and updated to version 6.0 in 2021 [9]. "ASN.1 is a formal notation used for describing data transmitted by telecommunications protocols" [12]. An ASN.1 file is organized in one or more data structures called module, each module contains one or more items. Such items are defined by standard that is composed of three main parts:

- The item name
- The characters "::="
- The type of the item.

Each module starts with its own name, followed by the header `DEFINITIONS AUTOMATIC TAGS ::=`. The beginning and the end of the module are marked by the keywords `BEGIN` and `END`, respectively. An example of module definition is shown below:

```
ModuleName DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

    ItemName1 ::= ItemType1
```

```
    ItemName2 ::= ItemType2

END
```

This is the easiest way to declare a module. ASN.1 offers the opportunity to add more information adding an *Object identifier* (OID), this OID can store different information, in the context of the ETSI ASN.1 files, this OIDs are used, to make an example, to specify the version of their respective modules. Below is listed an example from the official ETSI repository:

```
AVM-Commons {
  itu-t (0) identified-organization (4) etsi (0) itsDomain (5) wg1 (1)
  ts (103882) avmCommons (5) major-version-1 (1) minor-version-1(1)
}
DEFINITIONS AUTOMATIC TAGS ::= BEGIN  -- AVM-specific common data elements
```

2.1.1 ASN.1 Types

ASN.1 types can be divided into three main categories:

Basic Types These are the predefined basic types provided by ASN.1, used as building blocks for defining more complex structures. They include:

- **INTEGER** – Represents signed integer values.

  ```
  Number ::= INTEGER
  ```

- **BOOLEAN** – Represents a logical value, either `TRUE` or `FALSE`.

  ```
  Bool ::= BOOLEAN
  ```

- **IA5String** – Represents a string using the ASCII character set.

  ```
  ASCIIString ::= IA5String
  ```

- **UniversalString** – Represents a string using a 32-bit character representation for international character sets.

  ```
  UniString ::= UniversalString
  ```

- **BIT STRING** – Represents a sequence of individual bits, often used as bitmaps or flags.

```
Bit ::= BIT STRING
```

Constructed Types Using the basic types described above, it is possible to create more complex structures comprising more than one object.

- **SEQUENCE** – Represents a list of different item types. It has a fixed length and a predefined order.

```
Person ::= SEQUENCE {
    name IA5String,
    age  INTEGER
}
```

- **SEQUENCE OF** – Represents a list of items of the same type. It does not have a fixed length.

```
NameList ::= SEQUENCE OF IA5String
```

- **SET** – Represents a list of different item types. It has a fixed length, but the order is not relevant.

```
PersonSet ::= SET {
    name IA5String,
    age  INTEGER
}
```

- **CHOICE** – Represents a value that can be one of several alternative types. Only one alternative can be chosen when instancing the object.

```
Vehicle ::= CHOICE {
    car   IA5String,
    truck IA5String,
    bike  IA5String
}
```

Personalized Types ASN.1 allows the user to define new types based on the types defined above. These newly defined types can be used directly or as a component in the definition of more complex types.

```
Age ::= INTEGER (0..150)
```

```
Name ::= IA5String

Person ::= SEQUENCE {
    name Name,
    age  Age
}
```

2.1.2 ASN.1 Constraints

ASN.1 also allows the possibility to constraint the values that it's possible to assign to the object created using a specific type. The different type of constraint and how they work will be analyzed further in this project, at this point it is important to introduce what they are and how they work in principle. Taking as example the `INTERGER` type, in the ASN.1 standard is it possible to constrain its value to a single value, or to a range of values.

```
-- Range Constrain
AgeRange ::= INTEGER (0..150)
-- Single Value Constraint
AgeSingle ::= INTEGER (75)
```

One of the main strengths of how the constraint works in ASN.1 is that when a new type is defined using a constrained type, it will inherit all the constraints belonging to the parent type, this opens the path to the declarations of the new types used only declare the constraint on that single type, allowing the files to be more readable and modular.

2.1.3 ASN.1 Encoding Rules

ASN.1 not only specifies how the data structures are defined, but also describes the different encoding rules that can be used to transmit them. These encoding rules are described in ITU-T X 690-696 [12] and they are:

- **BER** (Basic Encoding Rules)
- **DER** (Distinguished Encoding Rules)
- **PER** (Packed Encoding Rules)
- **UPER** (Unaligned Packed Encoding Rules)
- **XER** (XML Encoding Rules)
- **OER** (Octet Encoding Rules)

The choice of which encoding rule to use depends on the context in which the ASN.1 format is used. For instance, the use of XER encoding rule facilitated the inspection of data during the debugging phase of this project. For the purpose of this work, the analysis of these rules and their application field is not relevant and will therefore be omitted.

2.1.4 ASN.1 Application Fields

Due to its versatility and standardizations, ASN.1 has been used in a variety of fields. Its first intended use was to specify the email protocol within the Open System Interconnection (OSI) environment [13]. Nowadays, its applications have expanded to a wide range of domains. It is employed for transmitting different types of messages over the internet, including audio and video, digital certificates and financial services. Both 3G and 4G technologies rely on ASN.1 for all the interactions between mobile devices and the carrier's network [13].
In the context of this thesis, ASN.1 is used in vehicular application to transmit messages described by ETSI C-ITS. However, the ASN.1 format has also been adopted in many others fields, from aviation or energy systems to security authentication and cryptography [13].

2.2 ASN.1 Compiler

The ASN.1 files discussed above only specifies the logical structure that the items described have and they are not ready for the use. These files need to be processed by dedicated software that parses the ASN.1 files and produces the corresponding implementation code for the chosen programming language. At the moment, there are commercial and open-source alternatives for such software. In this thesis, the focus is on the open source project *asn1c* which will serve as the basis for the new tool that will be developed; the other available solutions are listed below.

- **Commercial solutions**
  - *OSS ASN.1 Tools*, developed by OSS Nokalva [14]
  - *ASN1C Compiler*, developed by Objective Systems [15]
- **Open-source projects**
  - *asn1c*, the compiler used in this thesis, available on its GitHub page: `https://github.com/mouse07410/asn1c`
  - *ASN1SCC*, developed by the European Space Agency [16]
  - *asn1tools*, developed by Erik Moqvist and available on GitHub [17]
  - *Pycrate*, available on its GitHub page [18]

Other projects can be found on the dedicated page of ITU `https://www.itu.int/en/ITU-T/asn1/pages/tools.aspx`

2.2.1 asn1c Working Principle

To fully understand the themes of this work is essential to highlight how the *asn1c* software manages the .asn files. Its working flow can be divided into three main parts: **Parsing stage**, **Semantic Fixing** and **Code Generation**. Fig.2.1 The backbone of

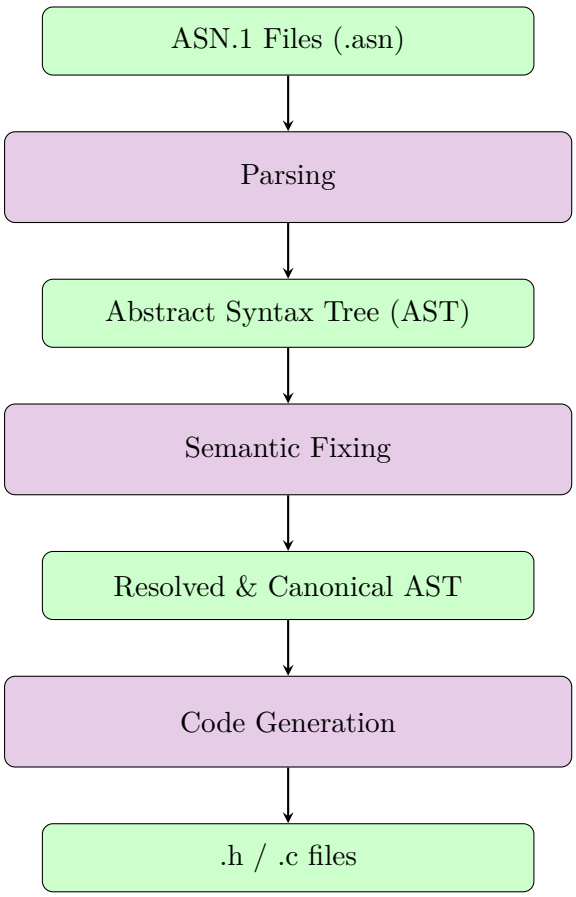

Figure 2.1. asn1c internal workflow: processes (violet) and isolated outputs (green).

the parsing stage is made using the open-source tool "GNU Bison". During this stage, the compiler reads the input ASN.1 files and produces an internal structure, called ASN.1 syntax tree (AST), which represents how they are understood by the compiler [19]. For debugging purposes, it is possible to print this intermediate representation using the command line option `-E`. An example is reported in the following. Given the ASN.1 file:

```
Shapes DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

Size ::= INTEGER (0..1000)
```

```
Rectangle ::= SEQUENCE {
    width   Size,
    height  Size OPTIONAL
}

END
```

The output using the command `asn1c -E` is:

```
Shapes DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

Size ::= INTEGER (0..1000)

Rectangle ::= SEQUENCE {
    width   Size,
    height  Size OPTIONAL
}

END
```

In this phase, the created AST is equal to the input file, since the purpose of the parser is only to read the files and to check if a particular construct is supported by the compiler [19]. After the parsing stage the AST is passed to the Semantic Fixing stage. During this stage the compiler performs a semantic normalization of the parsed structures, the tagging rules are applied, the constraints are recorded and saved in a dedicated structure that will later be used. The main purpose of this stage is to eliminate any kind of ambiguities and transform the parsed AST into a canonical representation which can be safely be used as the input for the code generation stage. As per the parsing stage it is possible to see part of the generated internal structures using the command line option `-EF`. An example is listed below.

The output using the command `asn1c -EF` is:

```
Shapes DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

Size ::= INTEGER (0..1000)

Rectangle ::= SEQUENCE {
    width   [0] IMPLICIT Size,
    height  [1] IMPLICIT Size OPTIONAL
}

END
```

In this example, the tags *[0], [1]* are generated by the compiler to unequivocally identify all the items inside the sequence.
The compiler final stage is the code generation stage. In this phase, the canonical AST is processed and all the necessary source code files are produced. The generated files includes both the C representations of the declared ASN.1 structures and the supporting functions that are required to encode and decode them, as well as to validate the data consistencies and the constraint validations.
This stage will be discussed in the following chapters, as the core of this thesis focuses on modifying the code generation component in order to implement the required changes.

Chapter 3

asn1c Problem Analysis

As mentioned different time in this thesis, the work is centered around the *asn1c* tool publicly available on the GitHub repository *mouse07410/asn1c*. The version employed in this work corresponds to the master branch, with the latest commit pushed on May 22, 2020 (commit hash 9925dbb).
In this chapter, the analysis is focused on the *asn1c* compiler with the objective of identifying its limitations. The adopted approach consist of two steps. Fist, a set of toy ASN.1 files have been designed and compiled in order to test the main functionalities of the tool in a controlled environment. This will allow a complete verification of the features that are correctly supported by the compiler. Second, the compiler is tested on real ASN.1 files provided by ETSI, which are know to produce compilation errors. The investigation of these errors makes it possible to understand the root causes of the failure and to highlight the limitations of the current build of the *asn1c* tool.

3.1 Constraint limitations

One of the most notable feature of the ASN.1 standard is the possibility to add constraint to a type. From the ITU X.680 definition, the ConstrainedType is defined as follow : "The "ConstrainedType" notation allows a constraint to be applied to a (parent) type, either to restrict its set of values to some subtype of the parent or (within a set or sequence type) to specify that component relations apply to values of the parent type and to values of some other component in the same set or sequence value" [9].
The real strength of the constrained type is that once the type is defined, all of its constrained are linked to it and all the new declared types that have the constrained one as parent will inherit all the constrains. This allow the ASN.1 files to be cleared and even more modular. For instance, is very common in ETSI's file to see at the begging of a module these declarations:

```
AVM-Commons {
  itu-t (0) identified-organization (4) etsi (0) itsDomain (5) wg1 (1)
  ts (103882) avmCommons (5) major-version-1 (1) minor-version-1(1)
```

```
}
DEFINITIONS AUTOMATIC TAGS ::= BEGIN

/**
 * This DE represents with an unsigned interger value.
 * Size: 8 bit, 1 Byte
 *
*/
UInt8 ::= INTEGER (0..255)

/**
 * This DE represents with a signed interger value.
 * Size: 16 bit, 2 Bytes
 *
*/
Int16 ::= INTEGER (-32768..32767)

[...]

END
```

These is a portion of the file *AVM-Commons.asn*, available on the ETSI official repository inside the folder containing the modules for Automated Vehicle Marshalling (AVM) [20]. In the reported example, the type shown represent an unsigned 8-bit integer and a signed 16-bit integer. Since the ASN.1 specifications do not define these types as native, the most efficient approach is to constrain the generic *INTEGER* using the so-called *Value Range* constraint, and then use the new defined types.
Given the importance constituted by the constraints, the first tests performed on the *asn1c* were about its ability on handle the different constraint types.

3.1.1 Subtype Element

One of the most common constraint types are the Subtype elements. The common feature of all these constraint is that they define a subset of the native type to which they are applied. As seen in the instance above, the *Value range* constraint defines a new subset of integer values that are contained between the lower and the upper bound.
The existing *Subtype elements* are described in the chapter 51 of the ITU X.680 and they are:

```
  SubtypeElements ::=
  SingleValue
| ContainedSubtype
| ValueRange
| PermittedAlphabet
| SizeConstraint
| TypeConstraint
| InnerTypeConstraints
| PatternConstraint
| PropertySettings
| DurationRange
| TimePointRange
| RecurrenceRange
```

Not all of them can be applied to every type, and below is provided a table, taken directly from the X.680 specification, in which are listed the different compatibilities between types and subtypes constraints [9].

Table 3.1. Applicability of "SubtypeElements" to types other than the Time type from [9]

Type (or derived from such a type by tagging or subtyping)	Single value	Contained subtype	Value range	Size constraint	Permitted alphabet	Type constraint	Inner subtyping	Pattern constraint
Bit string	Yes	No	No	Yes	No	No	Yes	No
Boolean	Yes	No	No	No	No	No	No	No
Choice	Yes	No	No	No	No	No	Yes	No
Embedded-pdv	Yes	No	No	No	No	No	Yes	No
Enumerated	Yes	No	No	No	No	No	No	No
External	Yes	No	No	No	No	No	Yes	No
Instance-of	Yes	No	No	No	No	No	No	No
Integer	Yes	No	Yes	No	No	No	No	No
Null	Yes	No	No	No	No	No	No	No
Object class field type	Yes	No	No	No	No	No	No	No
Object descriptor	Yes	No	No	Yes	Yes	No	No	No
Object identifier	Yes	No	No	No	No	No	No	No
Octet string	Yes	No	No	Yes	No	No	Yes	No
OID internationalized resource identifier	Yes	No	No	No	No	No	No	No
Open type	No	No	No	No	No	Yes	No	No
Real	Yes	No	Yes	No	No	No	Yes	No
Relative object identifier[b]	Yes	Yes	No	No	No	No	No	No
Relative OID intl. resource identifier[b]	Yes	Yes	No	No	No	No	No	No
Restricted character string types[a]	Yes	Yes	Yes[c]	Yes	Yes	No	No	Yes
Sequence	Yes	No	No	No	No	No	Yes	No
Sequence-of	Yes	No	No	Yes	No	No	Yes	No
Set	Yes	No	No	No	No	No	Yes	No
Set-of	Yes	No	No	Yes	No	No	Yes	No
GeneralizedTime and UTCTime	Yes	No	No	No	No	No	Yes	No
Unrestricted character string types	Yes	No	No	Yes	No	No	Yes	No

[a] Restricted character string types include `NumericString`, `PrintableString`, `IA5String`, `UTF8String`, etc.
[b] Relative object identifier and relative OID internationalized resource identifier accept both *Single value* and *Contained subtype*.
[c] Value range is applicable only to certain restricted character string types (e.g., `NumericString`).

Table 3.2. Applicability of "SubtypeElements" to the Time type from [9]

Type (or derived from such a type by tagging or subtyping)	Single value	Contained subtype	Property settings	Duration range	Time point range	Recurrence range	Inner subtyping
Time type	Yes	Yes	Yes	Yes	Yes	Yes	(Note)

NOTE – Only allowed if all the abstract values of the parent type have the property settings `"Basic-Interval Interval-type=D"` (see 38.4.4).

Before talking about the constraint that have been found without a counterparts in the code generation, it could helpful to better understand this work to have a look at what are we expecting too see as output.
Each module in the ASN.1 specification generates its own .c file, which contains all the functions and data structures required to represent and manage it. Below it is reported the function in charge of validating the constraint generated from the declaration of *Int16* shown above.

```
1  int
2  Int16_constraint(const asn_TYPE_descriptor_t *td, const void *sptr,
3              asn_app_constraint_failed_f *ctfailcb, void *app_key) {
4      long value;
5
6      if(!sptr) {
7          ASN__CTFAIL(app_key, td, sptr,
8              "%s: value not given (%s:%d)",
9              td->name, __FILE__, __LINE__);
10         return -1;
11     }
12
13     value = *(const long *)sptr;
14
15     if((value >= -32768L && value <= 32767L)) {
16         /* Constraint check succeeded */
17         return 0;
18     } else {
19         ASN__CTFAIL(app_key, td, sptr,
20             "%s: constraint failed (%s:%d)",
21             td->name, __FILE__, __LINE__);
22         return -1;
23     }
24 }
```

Listing 3.1. Constraint code validation for Int16 module

The created function in figure 3.1 has 4 arguments:

- `const asn_TYPE_descriptor_t *td`: Structure created by *asn1c* to represent the type and all its characteristics, in this example it is used to get the name of type (*td->name*)

- `const void *sptr`: Pointer to the value instance associated with the type. It is declared as void pointer to be more generic as possible and provide a valid interface for all the constraint functions. Within this function (line 13), it is cast to a pointer to long to access the actual integer value.

- `asn_app_constraint_failed_f *ctfailcb`: Application callback that is invoked whenever the constraint is violated or an error occurs. Its invocation is wrapped the `ASN__CTFAIL` macro, that standardized the call and manage the formatting and display of the error message.

- `void *app_key`: Pointer defined by the application. It is not interpreted by the constraint function, but it is used inside the `ASN__CTFAIL` macro.

The validation function for the *Value Range* constraint is simple but effective. It checks if the instanced value is contained in the range specified in the ASN.1 file, if the validation is successful the function return a positive feedback, otherwise if the constraint is not respected or the instanced value is void the function returns an error using the dedicated `ASN__CTFAIL` macro.
For brevity, in this thesis, only the constraint found to be lack of code implementation will be examined.

Single Value Constraint

The first constraint taken in exam is the single value constraint. This constraint restricts the possible values that a type can take to one or more values. One example of this constraint can be found inside the file "MSDASN1Module.asn" available in the ETSI repository: [21]

```
    MSDASN1Module DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

CurrentVersion ::= INTEGER (2)

[...]
END
```

This is an example of single value constraint applied to an integer type, in such example the new declared type "CurrentVersion" is limited to the value 2.
As it is possible to see in the table 3.1, the single value constraint can be applied also to others types, such as the strings type. The application of this constraint to these types is unusual e we were not able to find some official documents in which it is applied. Instead of using a sequence of single value constraint, it is more common to use the "Enumerated" type and provide a list of the possible value that the new type can take. Below an example, taken from the file "MSDASN1Module.asn", which shows how it is implemented.

```
    VehicleType ::= ENUMERATED {
   passengerVehicleClassM1(1),
   busesAndCoachesClassM2(2),
   busesAndCoachesClassM3(3),
   lightCommercialVehiclesClassN1(4),
   heavyDutyVehiclesClassN2(5),
   heavyDutyVehiclesClassN3(6),
   motorcyclesClassL1e(7),
   motorcyclesClassL2e(8),
   motorcyclesClassL3e(9),
   motorcyclesClassL4e(10),
   motorcyclesClassL5e(11),
   motorcyclesClassL6e(12),
   motorcyclesClassL7e(13)
   , ...
}
```

Despite being poorly used, since the constraint is present for the strings type, the capability of *asn1c* in generating the C code was tested. The example used for testing are taken from the quick start guide page of OSS Nokalva, one of the main supplier of ASN.1 compiler [22].

```
TextResponse ::= PrintableString ("Success" | "Failure")
WarningColors ::= IA5String ("Red" | "Yellow")
InfoColors    ::= UTF8String ("Blue" | "White")
```

In the case regarding the INTEGER single value constraint *asn1c* did generate the constraint check code, meanwhile for the strings the compiler did not generate any code functions to check if the restrictions are respected. Although, the use of the single value constraint is rare its implementation was decided to be carried out in order to improve the capacities of this new version of *asn1c*.

Pattern Constraint

Another other important constraint that was found without a C code implementation was the pattern constraint. The patter constrain it is applicable only to strings type and it declares a specific format that the string to which it is applied must follow. Below an example taken from the module Uds2 defined inside the ITU-T F.515 [23]:

```
NumericString-1 ::= IA5String(FROM ("0".."9"))(PATTERN "[0-9]")
```

Here the pattern constraint it is used to declare that the string will be only composed from the digit 0 to 9 and it is accepted only if it matches with the regular expression "[0-9]".

3.1.2 WITH COMPONENTS Constraint

One important constraint, that is widely used in the ETSI's ASN.1 file is the "WITH COMPONENTS" constraint. This constraint it is applicable on constructed types to add a more rigid restriction on the elements that compose the type. Below an example take from the ETSI's module "ETSI-ITS-CDD.asn" from CDD TS 102 894-2 [24].

```
    VruClusterInformation ::= SEQUENCE {
   clusterId                   Identifier1B OPTIONAL,
   clusterBoundingBoxShape     Shape (WITH COMPONENTS{...,
       elliptical ABSENT, radial ABSENT, radialShapes ABSENT}) OPTIONAL,
   clusterCardinalitySize      CardinalNumber1B,
   clusterProfiles             VruClusterProfiles OPTIONAL,
   ...
}

Shape::= CHOICE {
   rectangular       RectangularShape,
   circular          CircularShape,
   polygonal         PolygonalShape,
   elliptical        EllipticalShape,
   radial            RadialShape,
   radialShapes      RadialShapes,
   ...
}
```

In this example the WITH COMPONENTS it is used to specify that "clusterBoundingBoxShape" will inherit the Shape type but without the possibility to take the values "elliptical", "radial" nor "radialShapes". This gives the opportunity to declare a more generic type, like Shape, and then use it in different situations modifying which values it can assume depending from the context in which it is used.
Another use of the "WITH COMPONENTS" constraint is the follow:

```
    Schema DEFINITIONS AUTOMATIC TAGS ::= BEGIN

Address ::= SEQUENCE {
   street-address UTF8String,
   country UTF8String -- see a note below,
   postal-code UTF8String
} ((WITH COMPONENTS {
       ...,
       country ("USA"),
       postal-code (PATTERN "[0-9]#5(-[0-9]#4)?")
    }
  | WITH COMPONENTS {
       ...,
       country ("Canada"),
       postal-code (PATTERN "[0-9][A-Z][0-9]
                             [A-Z][0-9][A-Z]")
    }
  | WITH COMPONENTS {
       ...,
       country ("Netherlands"),
       postal-code (PATTERN "[0-9]#4 [A-Z]#2")
    }
  ))

END
```

This example, also taken from the site of OSS Novalka [22], shows how this constraint can also be use to apply new sets of constraints. In this specific case it is used to links the value that the type "country" can take to the relative "postal-code" value. Although the use of "WITH COMPONENTS", despite being flexible and with a significant space optimization capabilities, the alternative would have been to declare the three new different types based on the "Address" type, it is not employed in ETSI's ASN.1 files. Still its implementation is not complete and this feature is not present in *asn1c*.

3.2 Compilation Errors

The previous discussed missing features in *asn1c* where not blocking problems. The reason that move this thesis is the presence of some folder, inside the ETSI's official repository, that the actual version of the tool is not able to compile correctly. The main difference between the prior problems and the ones that will be discussed in this section is that in the first one *asn1c* does not generate the constraint validation code, but it does generate the all the core functions needed for the type representations and transmission. In the upcoming issues, *asn1c* is not able to generate the code at all for different reasons.

3.2.1 AVM TS 103 882 Error

At the execution of the command for generating the code from ETSI's folder "AVM TS 103 882" [20] a list of errors are shown.

Listing 3.2. Asn1c output when compiling the AVM TS files

```
...
FATAL: Cannot find external module "ETSI-ITS-CDD" mentioned for
   "VehicleMass" at line 263.
Obtain this module and instruct compiler to process it too.
in MVM-PDU-Descriptions.asn
...
```

The output displayed in 3.2 is only a part of the error logs, but since the error are all similar only this message is shown.
The error messages are clear, one or more modules are missing and for this reason *asn1c* is not able to correctly interpret the definitions. The problem in this case is that the module searched are effectively present inside the file but they are not recognized due to a error with the OID.

From MVV-PDU-Description.asn:

```
IMPORT
...
    ItsPduHeader, TimestampIts, VehicleMass, VelocityComponentValue
FROM ETSI-ITS-CDD {
  itu-t (0) identified-organization (4) etsi (0) itsDomain (5) wg1 (1)
  102894 cdd (2) major-version-4 (4) minor-version-1 (1)
} -- WITH SUCCESSORS
;
```

From ETSI-ITS-CDD.asn:

```
    ETSI-ITS-CDD {itu-t (0) identified-organization (4) etsi (0)
    itsDomain (5) wg1 (1) 102894 cdd (2) major-version-4 (4)
    minor-version-2 (2)}

DEFINITIONS AUTOMATIC TAGS ::=

BEGIN
...
    VehicleMass ::= INTEGER {
    outOfRange (1023),
    unavailable(1024)
} (1..1024)
...
END
```

As it shown above, in both the importer and exporter, the type "VehicleMass" is correctly define. The issue in this case is caused by a missing interpretation of the tag *WITH SUCCESSORS*. This tag should give the information to *asn1c* that the accepted module are not only the one with the exactly OID, but also the one with a version greater or equal that the one declared in the import section.

3.2.2 IEEE 1609.2.1 Error

When compiling the IEEE 1609.2.1 folder from the ETSI's repository another error occurs, different the one discussed above.

Listing 3.3. Asn1c output when compiling the IEEE 1609.2.1 files

```
...
FATAL: Type ScmsPdu-Scoped expects specialization from ScmsPdu-
   Scoped at line 448 in Ieee1609Dot2Dot1Protocol.asn
FATAL: Cannot find type "<Type>" in constraints at line 452 in
   Ieee1609Dot2Dot1Protocol.asn
FATAL: Type ScmsPdu-Scoped expects specialization from ScmsPdu-
   Scoped at line 453 in Ieee1609Dot2Dot1Protocol.asn
FATAL: Cannot find type "<Type>" in constraints at line 457 in
   Ieee1609Dot2Dot1Protocol.asn
FATAL: Type ScmsPdu-Scoped expects specialization from ScmsPdu-
   Scoped at line 458 in Ieee1609Dot2Dot1Protocol.asn
FATAL: Cannot find type "<Type>" in constraints at line 462 in
   Ieee1609Dot2Dot1Protocol.asn
FATAL: Type ScmsPdu-Scoped expects specialization from ScmsPdu-
   Scoped at line 463 in Ieee1609Dot2Dot1Protocol.asn
FATAL: Cannot find type "<Type>" in constraints at line 468 in
   Ieee1609Dot2Dot1Protocol.asn
...
```

In this case, the error is caused because *asn1c* is not able to fully understand the structure present in the file "Ieee1609Dot2Dot1Protocol.asn" and this causes a series of error that prevent the code from being generated.
The section of the ASN.1 file that causes the problem is the following:

```
    ScopedCertificateRequest ::= ScmsPdu (
  ScmsPdu-Scoped {
    AcaRaInterfacePdu (WITH COMPONENTS {
      raAcaCertRequest
    })
  } |
  ScmsPdu-Scoped {
    EcaEeInterfacePdu (WITH COMPONENTS {
      eeEcaCertRequest
    })
  } |
  ScmsPdu-Scoped {
    EeRaInterfacePdu (WITH COMPONENTS {
      eeRaCertRequest
    })
  } |
  ScmsPdu-Scoped {
    EeRaInterfacePdu (WITH COMPONENTS {
      eeRaSuccessorEnrollmentCertRequest
    })
  }
)
```

The error caused from this file is a known limit of *asn1c* and it is reported in the issue #225 of the official repository of *asn1c*. For this problem there are solutions around, like the one proposed by *vanetza* team in the pull request #223 of their project [25].
This solution, despite it works, it only a workaround that does not fix the problem. Instead of working directly on *asn1c*, they modified the ASN.1 files that causes the issue not changing its meaning. This solution was initially considered but in the end it was discarded because, in case of new ASN.1 files with a similar structure, the compiler would fail again.

3.3 Error Causes

Once that all the problems are listed, the next step is to investigate the causes behind such problems.

3.3.1 Constraint Limitations

As mentioned before, all the issue related to the constraint problem are not blocking problem since the *asn1c* tool is able to generate all the essential functions. The success of the compiling procedure can give as an hint on where the problem is situated. As described in figure 2.1, the compiling process can be divided in three steps, if one between

the parsing or the fixing stage fails, the entire process fail, so it is reasonable to search the problems' root inside the part that is in charge of the code generation. Luckily the *asn1c* project is well divided and all the functions related to the code generation are inside its folder "libasn1compiler". Inside this folder there are all the *asn1c* functions related to the code generation.
Inside this folder we can find the file "asn1c_constraint.c", given the file name it was the first file that was inspected to understand where the constraint code generations fails. Inside this file is present the function "asn1c_emit_constraint_checking_code" that is the one in charge of generate the C code for the constraint. Inspecting the code it is clearly visible that there are no functions that handle the constraint discussed above. A possible solution, that will be discussed in deep in the next chapter, is to write the constraint generation code following the example of the already existing one to create an homogeneous final C code.

3.3.2 AVM TS 103 882

After a careful analysis of the error presented in 3.2, the reason of the error can be possible be found in the "WITH SUCCESSOR" tag. This tag is used in the "IMPORT" section of the module "MVV-PDU-Description.asn" and its purpose it's to declare that the module accepts the required module, in this case "ETSI-ITS-CDD", with the required version or a more recent one. In this case the request version is the 4.1 [3.2.1] and the reported version in "ETSI-ITS-CDD.asn" file is 4.2 that should be supported. The solution that is implemented in this thesis is to delegate the version check to other functions, improving also the log messages to provide a better user experience.

3.3.3 IEEE 1609.2.1

Since the issue withe the IEEE 1609.2.1 files is known, it is possible to have a more detailed explanation on its causes. For the code maintainer, the problem is related to the complexity of the structures cited in 3.2.2. Such complexity generate various error when the tool tries to include the type header in the right order inside the its own data structures. The proposed solution in this thesis aims to reduce the complexity of such structures so that *asn1c* can properly processes them.

Chapter 4

Code Implementation

Given the issues presented in chapter 3, the adopted solutions for such issues will be discussed in this chapter. The proposed solution will be divided in two parts, one for each problem type that was previously discussed.

4.1 Constraint Validation

In this section, the code implemented to allow the correct constraint validation will be discussed. All the solutions here proposed are implemented inside the file "asn1c_constraint.c" which is the responsible for generate the code related to the constraint validation.// Before exploring the newly implemented functions, it is crucial to understand how *asn1c* handles the code generation and how it represent internally the ASN.1 structures, as this knowledge constitutes the foundation for the implementation of the proposed code.

4.1.1 *asn1c* Tools

The most important tools given by *asn1c* are the macros defined inside the file *ans1c_out.h*. These macros are used to help invoke the function *asn1c_compiled_output* defined in the file *asn1c_out.c*. This function takes as input a string and a pointer referencing the current position in the output stream. All the macros here presented are realized with the aim to facilitate the use of the above mentioned function, here some of the most used:

- `OUT`: This is the most used macro in the project, it allows to print in the output stream easily, it will accept a string as input and it will pass all the needed arguments to the *asn1c_compiled_output*.
- `INDENT`: This macro is mainly used to control the indentation level of the generated code. It accepts positive integer to increase indentation, negative to decrease it.
- `REDIR`: This macro is used to redirect the pointer that references to the current position in the output stream. Several redirection point, that can be called inside this macro, are defined in the code to easily move the output stream to specific sections of the.

- `GEN_INCLUDE`: This macro is used to generate the library include area. It takes as input the desired library and it will redirect the output stream to the library area, print the code related to the library, and come back to the original output stream point.

Another important aspect to present before exploring the proposed solutions is the internal representation of the constraint. For each type is created a struct called *asn1p_constraint_t* which contains the relevant information about the constraint and its surrounding context. Below it is shown an example of such representation from the pattern constraint presented in chapter 3.
The constraint structure is defined as a tree structure. For each type when a constraint is present the root element is created, this root is always defined as a **ACT_CA_SET**. The set is always created, even if only a single constraint is present. The information about the real constraint are stored inside the elements array. This is an array in which all the constrain applied to the type are listed, in the parent constraint is also listed the number of elements under the field *el_count*. Another important filed is the *module* field. In this field are stored all the data about the current module in which the type is defined such as the module name and path, the module which is going to be imported or exported and other information not relevant for the scope of this thesis.
One notable detail of this structure is the presence of several empty field. Despite this, the structure operates correctly, allowing a high level of modularity and enabling the representation of constraint with different characteristic using the same structure. In box 4.1 it shown the root constraint generated for this example. After presenting the structure of the root constraint, below it is presented the structure one of the elements presents in the structure above. Several difference can be spotted, the first one is the presence of pointer referencing to the parent constraint. The other main difference is that, in this case, the *value* field is populated with the string contained in the pattern defined in the ASN.1 file.
In box 4.2 is shown the structure taken from the pattern constraint discussed above.

Listing 4.1. Constraint root ct

```
asn1p_constraint_t *ct  (0x555555622140)
|-- type               = ACT_CA_SET
|-- presence           = ACPRES_DEFAULT
|-- parent_ct          = NULL
|-- containedSubtype   = (asn1p_value_t *) NULL
|-- value              = (asn1p_value_t *) NULL
|-- range_start        = (asn1p_value_t *) NULL
|-- range_stop         = (asn1p_value_t *) NULL
|-- elements           = (asn1p_constraint_t **) 0x555555622350
|   |-- [0] -> asn1p_constraint_t *result (see below)
|   \-- [1] -> asn1p_constraint_t *...
|-- el_count           = 2
|-- el_size            = 4
|-- module             = (asn1p_module_s *) 0x555555620680
|   |-- ModuleName          = "Uds2"
|   |-- source_file_name    = "preprocessed_temp/Uds2.asn"
|   |-- module_oid          = NULL
|   |-- module_flags        = MSF_AUTOMATIC_TAGS
|   |-- exports             = {...}
|   |-- imports             = {...}
|   |-- members             = {...}
|   |-- members_hash        = 0x5555556624a0
|   |-- mod_next            = {...}
|   |-- asn1p               = 0x555555678090
|   |-- _tags               = 0
|   \-- _lineno             = 54
|-- _lineno            = 3
\-- pattern_data       = (void *) NULL
```

Listing 4.2. Child constraint result (PATTERN)

```
asn1p_constraint_t *result  (0x555555622380)
|-- type              = ACT_CT_PATTERN
|-- presence          = ACPRES_DEFAULT
|-- parent_ct         = (asn1p_constraint_t *) 0x555555622140  -> ct
|-- containedSubtype  = (asn1p_value_t *) NULL
|-- value             = (asn1p_value_t *) 0x5555556223e0
|   |-- type               = ATV_STRING
|   \-- value (union)
|      |-- constraint         = (asn1p_constraint_s *) 0x555555561e820
|      |-- v_type             = (asn1p_expr_s *)       0x555555561e820
|      |-- reference          = (asn1p_ref_t *)        0x555555562e0
|      |-- v_integer          = 9223338411933540818976
|      |-- v_double           = 4.6355705792682058e-310
|      |-- string
|      |   |-- buf  = (uint8_t *) 0x555555561e820 "[0-9]"
|      |   \-- size = 5
|      |-- binary_vector       = {...}
|      \-- choice_identifier   = {...}
|-- range_start       = (asn1p_value_t *) NULL
|-- range_stop        = (asn1p_value_t *) NULL
|-- elements          = (asn1p_constraint_t **) NULL
|-- el_count          = 0
|-- el_size           = 0
|-- module            = (asn1p_module_s *) 0x555555620680
|-- _lineno           = 3
\-- pattern_data      = (void *) NULL
```

4.1.2 Code Generation

With all the necessary background introduced, the implementation code is now presented. The first modification concerns how the constraint are accessed, inside the *asn1c_emit_constraint_checking_code*, after the existing code validation part, a simple but functional code was designed to access all the different constraint.

```
1  if (ct->type == ACT_CA_SET && ct->elements != NULL) {
2          for (unsigned int i = 0; i < ct->el_count; i++) {
3              // Pattern constraint implementation
4              if(ct->elements[i]->type == ACT_CT_PATTERN) {
5                  emit_regex_include(arg);
6                  emit_pattern_constraint(arg, ct, i);
7              }
8
9              // Single value string constraint implementation
10             if(ct->elements[i]->type == ACT_EL_VALUE && etype &
       ASN_STRING_MASK) {
11                 emit_regex_include(arg);
12                 emit_single_value_string_constraint(arg, ct,i);
13             }
14            ...
15             if(ct->elements[i]->type == ACT_CA_UNI) {
16                 // Union constraint handling
17             ...
18             }
19         }
20     }
```

Listing 4.3. Constraint access code

This code is based on the principle that every constraints representation has a root set constraint that contains all of them as described above. Exploiting this principle, with a simple iteration over all the elements of the set, is it possible to access all constraint.// As shown in Listing 4.3, the proposed implementation concerns the pattern constraint and the single value constraint for the string types. On notable detail is that, for the single value constraint, in addition to checking the constraint type, an additional check is performed on the element type. The variable *etype* stores the type of the object to which the constraint is applied, and is it possible to perform a bitwise AND operation with a mask containing the string type information.
At the end there is also a space for the so called *Union constraint*, this constraint will be discussed later.

Single Value Constraint

The first constraint that will be discussed is the *Single value constraint*. The code implementation is shown in Listing 4.4. The working principle is simple but functional, the first step is to obtain the string value contained in the constraint. Before using it, the obtained string needs to be processed by new function *escape_for_c_string*. This function receives a string as input and inserts the appropriate escape sequences, ensuring that the resulting string can be safely handles by the generated C code.
Once the string is correctly managed the function generated the code that is in charge for the constraint validation. This operated is managed by using the *strcmp* function, comparing the escaped string with the actual value of the string contained in the object. The structure that manage the error (lines 14-18) is a structure native of *asn1c* that here

is used only changing the error message that should be displayed in case of error.

```
1  static void
2 emit_single_value_string_constraint(arg_t *arg, asn1p_constraint_t
      *ct, int i) {
3      if(ct->elements[i]->value->value.string.buf != NULL) {
4          const char *raw_constraint_value = (const char
      *)ct->elements[i]->value->value.string.buf;
5          char *escaped_constraint_value =
      escape_for_c_string(raw_constraint_value);
6
7          // Field name for error messages (if available)
8          const char *field_name_for_error = (arg->expr &&
      arg->expr->Identifier) ? arg->expr->Identifier : "field";
9
10         // Generate C code for validation
11         OUT("            char *actual_runtime_value = strndup((const
      char *)st->buf, st->size);\n");
12         OUT("            if(!actual_runtime_value) {\n");
13         INDENT(+1);
14         OUT("                ASN__CTFAIL(app_key, td, sptr, \"%%%%s:
      strndup failed for component '%s' (%%%%s:%%%%d)\", td->name,
      __FILE__, __LINE__);\n", field_name_for_error);
15
16         OUT("                return -1;\n");
17         INDENT(-1);
18         OUT("            }\n");
19
20         if (escaped_constraint_value) {
21             OUT("            const char *expected_constraint_literal
      = \"%s\";\n", escaped_constraint_value);
22             OUT("            if (strcmp(actual_runtime_value,
      expected_constraint_literal) != 0) {\n");
23             INDENT(+1);
24             OUT("                ASN__CTFAIL(app_key, td, sptr,
      \"%%%%s: component '%s' value ('%%s') does not match constraint
      '%%s' (%%%%s:%%%%d)\",\n", field_name_for_error);
25             OUT("                    td->name, actual_runtime_value,
      expected_constraint_literal, __FILE__, __LINE__);\n");
26             OUT("                free(actual_runtime_value);\n");
27             OUT("                return -1;\n");
28             INDENT(-1);
29             OUT("            }\n");
30             OUT("            free(actual_runtime_value);\n");
31         } else {
32             // If there was an error escaping the constraint value,
      this is an asn1c internal issue.
33             // Generate code to free actual_runtime_value and fail.
34             OUT("            ASN__CTFAIL(app_key, td, sptr, \"%%%%s:
      internal error escaping constraint value for component '%s'
      (%%%%s:%%%%d)\", td->name, __FILE__, __LINE__);\n",
      field_name_for_error);
35             OUT("            free(actual_runtime_value);\n");
```

```
36              OUT("            return -1;\n");
37          }
38
39          // Free the memory allocated by escape_for_c_string in this
        C function (emit_single_value_string_constraint)
40          if(escaped_constraint_value) {
41              free(escaped_constraint_value);
42          }
43      }
44  }
```

Listing 4.4. Single value constraint generation code

Pattern Constraint

The other constraint whose code generation will be discussed in this thesis work is the patter constraint. The pattern validation mechanism is based on one the most established concepts in software engineering, the validation and interpretation of regular expression. Since this is a well know problem different libraries offers their tools to handle it. During this work two different libraries were taken under the lens to see which of them could better fit the needs of this application. The two libraries are the *regex.h* library and the *pcre2.h* library.
Both libraries provide similar functionalities, one of main differences between the two libraries is the syntax that they could support. The *regex.h* library is based on the standard *POSIX* (Portable Operating System Interface) [26] first defined in the ISO 9945-2:1993 [27], this standard is the foundation of how the regular expression pattern are written. Meanwhile the *pcre2.h* library supports the *POSIX* standard but it also add some special characters that extend it, such as the "$\backslash d$, $\backslash w$" and many other that are also used by the ASN.1 standard [9]. Despite being able to fully support the special character described by in the standard ITU X680, and also being notoriously more efficient than is counterpart, the library *pcre2.h* has been discard in favor of *regex.h* for compatibility reason.
The library *regex.h* is one of the library available in the C standard library group while *pcre2.h* is not. Since *asn1c* is a code generator tool, it is not known in which ambient the generated code will be compiled and for such reason it is unknown if is in such ambient the *pcre2.h* library is installed. Moreover, since the pattern constraint is not commonly present in all the ASN.1 files, it would not be considerable practical to impose the installation of an additional library that they will may not use.
After established which library will be used in this project, the code implementation can be discussed. The proposed solution listed below in List 4.5, in this solution the regular expression is evaluated using the *regex.h* functions. As for the single value constraint, a function for handling the pattern string is created. To handle the lack of functionalities presents in the *regex.h* library the original function *escape_for_c_string* was modified to cover some of the most used special characters. The supported charters on this version of the project are:

- `\d`: This character will generate the equivalent in *POSIX* [0-9]
- `\w`: This character will generate the equivalent in *POSIX* [a-zA-Z0-9]
- `#n`: This character will generate the equivalent in *POSIX* { n } where n is an integer

```
1  static void
2  emit_pattern_constraint(arg_t *arg, asn1p_constraint_t *ct, int i) {
3      if(ct->elements[i]->value->value.string.buf != NULL) {
4
5          OUT("const char *c_string = strndup((const char *)st->buf,
       st->size);\n");
6          const char *pattern = (const char
       *)ct->elements[i]->value->value.string.buf;
7          char *escaped_pattern = escape_for_c_string_pattern(pattern);
8
9          if (escaped_pattern) {
10             OUT("const char *string_pattern =  \"%s\";\n",
       escaped_pattern);
11             free(escaped_pattern); // Free memory after use
12         } else {
13             // If escaping fails, use the original pattern but it
       may not work
14             OUT("const char *string_pattern =  \"%s\";\n", pattern);
15         }
16
17         OUT("regex_t regex;\n");
18         OUT("int ret = regcomp(&regex, string_pattern ,
       REG_EXTENDED);\n");
19         OUT("if (ret) {\n");
20         OUT("ASN__CTFAIL(app_key, td, sptr,\n");
21         OUT("\t\"%%s: constraint failed (%%s:%%d)\",\n");
22         OUT("\ttd->name, __FILE__, __LINE__);\n");
23         OUT("return -1;\n");
24         OUT("    return -1;\n");
25         OUT("}\n");
26         OUT("\n");
27         OUT("ret = regexec(&regex, c_string, 0, NULL, 0);\n");
28         OUT("regfree(&regex);\n");
29         OUT("if (ret) return -1;\n");
30     }
31 }
```

Listing 4.5. Patter constraint generation code

Union Constraint

As stated multiple times during this work, one of the major strength point of ASN.1 it is its modularity. When applying a constraint to a type, ASN.1 allows applying more than one constraint to the same type combining their value through a logical union constraint.

Since the constraint created is a different type of constraint, called *union* constraint, it is not recognized by the solution proposed above. To overcome this problem a solution, two new different functions have been designed to implement the missing functionalities. The proposed implementation is shown in List 4.6. The *union* constraint stores the information in the same way the *set* constrain does and the solution adopts a similar structure to the one described before, iterating over all the elements of the constraint and generating the corresponding code.

```
1  ...
2  if(ct->elements[i]->type == ACT_CA_UNI) {
3                  // Union constraint handling
4
5                  if(ct->elements[i]->el_count > 0) {
6                      OUT("int union_contains = 0;\n");
7                  }
8                  for (unsigned int j = 0; j <
       ct->elements[i]->el_count; j++) {
9
10
11                     if(ct->elements[i]->elements[j]->type ==
       ACT_EL_VALUE && etype & ASN_STRING_MASK) {
12                         if (value_found == 0) {
13                             value_found = 1;
14                             emit_regex_include(arg);
15                         }
16
       emit_single_value_string_constraint_union(arg, ct,i,j,
       first_string);
17                         first_string++;
18                     }
19
20                     if(ct->elements[i]->elements[j]->type ==
       ACT_CT_PATTERN) {
21                         // printf("DEBUG\n");
22                         if (value_found == 0) {
23                             value_found = 1;
24                             emit_regex_include(arg);
25                         }
26                         emit_pattern_constraint_union(arg, ct,i,j,
       first_pattern);
27                         first_pattern++;
28                     }
29                 }
30                 if (value_found) {
31                     OUT("if (union_contains == 0) {\n");
32                     INDENT(+1);
33                     OUT("ASN__CTFAIL(app_key, td, sptr,\n");
34                     OUT("\t\"%%s: constraint failed (%%s:%%d)\",\n");
35                     OUT("\ttd->name, __FILE__, __LINE__);\n");
36                     OUT("return -1;\n");
37                     INDENT(-1);
38                     OUT("}\n");
39                 }
40             }
41             ...
```

Listing 4.6. Union constraint generation code

4.2 Files Preprocessing

After discussing the implementation of constraint handling, the remaining issues that will be discussed in this work are related to the structure of the ASN.1 files themselves. One possible solution to solve the errors presented in section 3.2.1 and 3.2.2 could have been implemented modifying the core structure of *asn1c* that manages the creation of AST. This alternative would have required a deeper analysis of the project and consequently more time to implement the solution, not to the mention the potential bugs that could arises when modifying a project of such complexity. To avoid this scenario and to keep the complexity of this project as low as possible, another solution, openly inspired by project *vanetza*, an "open-source implementation of the ETSI C-ITS protocol" [28] stack freely available on its github repository at `https://github.com/riebl/vanetza`.
In project *vanetza* the IEEE 1609.2.1 ASN.1 files are modified before using them. This solution was not directly taken in consideration because only manages the nominated files and not solve the cause of the problem that could happens for other ASN.1 files. From this idea, the proposed solution was created, consisting of adding a preprocessing layer to *asn1c*. This newly defined layer will be responsible for both managing the version control, that cause the problem seen in 3.2.1, and simplifying the complex structures that cause the errors in 3.2.2.

4.2.1 Version Control Preprocessing

As described in 3.2.1, this specific problem is related to a missing interpretation of the tag *WITH SUCCESSOR*. The proposed solution to this problem consists in replacing the role of *asn1c* with a dedicated script that will check the compatibility.
This algorithm is implemented in the file *import.c*. First, the algorithm search and store inside the memory for each ASN.1 file analyzed the module's name and its OID and, if they exist, the requested modules' name and OIDs. This process is made easier by the structure of the ASN.1 files themselves. In every ASN.1 file the module name and its OID are always the first meaningful words in the file. For what regards the modules that are requested, the rigid structure of ASN.1 files again help to gather this info. Such modules are always declared in the first part of the file and are declared between the word *IMPORTS* and the character *;*. In this section are defined the requested types followed by the word *FROM* which specifies the module's name ad OID. Contextually with the information gathering process, the scripts is also responsible for removing the OIDs from the ASN.1 files, preventing *asn1c* from generating errors caused by different OIDs.
Once all the modules information are available, the program will compare the requested modules version with the ones available, and if there is a positive match it will pass the ASN.1 files to the next step of the preprocessing layer that will be discussed in the next section.
In case of negative results an error message will be shown but the process is not stopped, the user will be notified with which modules could possibly cause an error but the choice if continue or not will be taken by him. This choice was made because the declaration of

the OID is not mandatory in ASN.1 standard. In any case, if the user choices to continue even if the requested modules are not present, *asn1c* will generate an error similar to the one presented in 3.2.1, but this time caused by a real case of missing module or type.
To consult the complete code with all the commented functions, refer to the public repository in which the project is uploaded.

4.2.2 WITH COMPONENTS Preprocessing

Before presenting the implemented solution that allows *asn1c* to handle the IEEE 1609.2.1 ASN.1 files, the approach adopted by *vanetza* project will be analyzed, since it represent the starting point for the proposed implementation.
The team behind *vanetza* decided to manually modify the ASN.1 files that generate the error described in 3.2.2. Their patch can be found in their repository under this path `vanetza/asn1/patches/ieee/Ieeedot2dot1Protocol.patch`. The patch, available below, takes the complex structure of the type *ScopedCertificateRequest*, that can assume four different types each of them constrained by a *WITH COMPONENTS* constraint, and split it in two separate part. First, it will declare the four different constrained types in a separate section. Second, it will replace the existing structure with only the four previously declared types.
Here it is reported the full patch:

```
+ScmsPdu-RaAcaCertRequest ::= ScmsPdu-Scoped {
+  AcaRaInterfacePdu (WITH COMPONENTS {
+    raAcaCertRequest
+  })
+}
+ScmsPdu-EeEcaCertRequest ::= ScmsPdu-Scoped {
+  EcaEeInterfacePdu (WITH COMPONENTS {
+    eeEcaCertRequest
+  })
+}
+ScmsPdu-EeRaCertRequest ::= ScmsPdu-Scoped {
+  EeRaInterfacePdu (WITH COMPONENTS {
+    eeRaCertRequest
+  })
+}
+ScmsPdu-EeRaSuccessorEnrollmentCertRequest ::= ScmsPdu-Scoped {
+  EeRaInterfacePdu (WITH COMPONENTS {
+    eeRaSuccessorEnrollmentCertRequest
+  })
+}
```

```
 ScopedCertificateRequest ::= ScmsPdu (
-  ScmsPdu-Scoped {
-    AcaRaInterfacePdu (WITH COMPONENTS {
-      raAcaCertRequest
-    })
-  } |
-  ScmsPdu-Scoped {
-    EcaEeInterfacePdu (WITH COMPONENTS {
-      eeEcaCertRequest
-    })
-  } |
-  ScmsPdu-Scoped {
-    EeRaInterfacePdu (WITH COMPONENTS {
-      eeRaCertRequest
-    })
-  } |
-  ScmsPdu-Scoped {
-    EeRaInterfacePdu (WITH COMPONENTS {
-      eeRaSuccessorEnrollmentCertRequest
-    })
-  }
+ScmsPdu-RaAcaCertRequest | ScmsPdu-EeEcaCertRequest |
+ScmsPdu-EeRaCertRequest  | ScmsPdu-EeRaSuccessorEnrollmentCertRequest
 )

 /**
```

This approach was improved and was designed to be compatible with all the ASN.1 files. Since the main problem is given by the complexity of the structures, the idea is to create an algorithm capable of finding such complex in every ASN.1 files givens as input and, if found, substitute such structure with two different section like the patch analyzed above did.
First, each file is analyzed individually. The research for this structure is divided in two part. First, the algorithm search for every type declaration inside the ASN.1, then for each type a control is performed to check if in it there is a structure similar to the one discussed before. To be more precise, the script search for a structure of this type:

```
    MyChoiceType ::= CHOICE (
  scoped-type-A { inner-type-A (WITH COMPONENTS { component-1 }) } |
  scoped-type-B { inner-type-B (WITH COMPONENTS { component-2 }) } |
  scoped-type-C { inner-type-C (WITH COMPONENTS { component-3 }) }
)
```

If this structure is found the second phase of the algorithm is executed. In this section,

the newly found structure is deleted and replaced with an easier structure that now is correctly managed by *asn1c*. The results will have this structure:

```
    -- Scoped type aliases, generated automatically
AliasComponent-1 ::= scoped-type-A {
  inner-type-A (WITH COMPONENTS {
    component-1
  })
}

AliasComponent-2 ::= scoped-type-B {
  inner-type-B (WITH COMPONENTS {
    component-2
  })
}

AliasComponent-3 ::= scoped-type-C {
  inner-type-C (WITH COMPONENTS {
    component-3
  })
}

MyChoiceType ::= CHOICE (
  AliasComponent-1 |
  AliasComponent-2 |
  AliasComponent-3
)
```

This approach, unlike the one proposed by *vanetza*, not only fix the issue with IEEE 1609.2.1 files, but also guarantees that *asn1c* can handle such complex structures autonomously, allowing for a broader range of use and completely eliminating the need for human intervention.
As for the version control algorithm, the user is constantly kept informed through debug messages about the operations performed by the different functions and whether the script had to intervene to modify the ASN.1 files.
To consult the complete code with all the commented functions, refer to the file *prepocessor.c* available in the repository in which the project is uploaded.

4.3 Final Code Implementation

In the first draft of this work, the preprocessing layer was designed as a completely separate C program to be executed before the use of *asn1c*. The reason behind this choice was to reduce modifications to the original software while still providing the tools to enable it to handle the newly released files. However, during the testing phase the clear

division between the preprocessing layer and *asn1c* turned out to be more a downside than an advantage. It was decided to fully integrate this new layer to *asn1c*, to do so the main function was modified including a call to the file *preprocessor.c*. As mentioned earlier, this file was initially designed to be executed as a standalone rather than being called as a function by another main routine so there was the need to be slightly modified to be compatible with *asn1c*.
The only addition that was made was the adding of another tag to *asn1c*. Natively *asn1c* supports different tag for that enables different debug functionalities all listed in the official documentation. The newly added tag is *-Wdebug-pre*, this tag enable a more verbose output, providing detailed information about the operation performed by all the preprocessing layer.

4.4 Final Test and Code Generation

Once the implementation phase was completed, different test were performed. The first set of test was about the implementation of the constraint generation. In addition to test performed on toy examples during the developing phase, the entire project was tested with the ASN.1 files mentioned in 3.1. All the performed tests produce a positive outcome and the expected constrain code was successfully generated. Below the results for single value constraint 4.7 and for pattern constraint 4.8.
The second set of test were performed directly on all the available file on the official ETSI repository. These tests have a double scope, the first is to check if the added preprocessing layer worked as intended, allowing *asn1c* to correctly generate the C code for the ASN.1 files discussed in 3.2 . The second purpose of the test session was to ensure that the other folders, that were correctly interpreted by the base version of the software, are still correctly managed. Also for these tests, the outcome was successful for all the folders under the analysis.
In the end, all the meaning full tests about the code generation were performed and all of them had a positive results. A last test was performed on IEEE 1609.2.1 files to check if the code its actually able to decode a real packet sent by a vehicle. Using the generated libraries, a simple code that decode a real packet has been designed. Below the function that implement the decoding function.
The function in 4.9 rely on the tools provided by *asn1c*. The principal tool is the function *ber_decode* that is able to decode, from a byte string encoded using the ber protocol, the bite in input.
This last test provided a positive feedback, proving that the proposed solution effectively solves the initial problem that motivated the thesis project.

```
1  int
2  TextResponse_constraint(const asn_TYPE_descriptor_t *td, const void
       *sptr,
3              asn_app_constraint_failed_f *ctfailcb, void *app_key) {
4      const PrintableString_t *st = (const PrintableString_t *)sptr;
5
6      if(!sptr) {
7          ASN__CTFAIL(app_key, td, sptr,
8              "%s: value not given (%s:%d)",
9              td->name, __FILE__, __LINE__);
10         return -1;
11     }
12
13     int union_contains = 0;
14     const char *c_string = strndup((const char *)st->buf, st->size);
15     char *single_value =  "Success";
16     if (strcmp(c_string, single_value) == 0) {
17               union_contains = 1;
18         }
19      single_value =  "Failure";
20     if (strcmp(c_string, single_value) == 0) {
21               union_contains = 1;
22         }
23     if (union_contains == 0) {
24         ASN__CTFAIL(app_key, td, sptr,
25             "%s: constraint failed (%s:%d)",
26             td->name, __FILE__, __LINE__);
27         return -1;
28     }
29
30     ...
31 }
```

Listing 4.7. Generated code for Single value constraint

```
1  int
2  NumericString_1_constraint(const asn_TYPE_descriptor_t *td, const
       void *sptr,
3              asn_app_constraint_failed_f *ctfailcb, void *app_key) {
4      const IA5String_t *st = (const IA5String_t *)sptr;
5
6      if(!sptr) {
7          ASN__CTFAIL(app_key, td, sptr,
8              "%s: value not given (%s:%d)",
9              td->name, __FILE__, __LINE__);
10         return -1;
11     }
12
13     const char *c_string = strndup((const char *)st->buf, st->size);
14     const char *string_pattern =  "[0-9]";
15     regex_t regex;
16     int ret = regcomp(&regex, string_pattern , REG_EXTENDED);
17     if (ret) {
18         return -1;
19     }
20
21     ret = regexec(&regex, c_string, 0, NULL, 0);
22     regfree(&regex);
23     if (ret) return -1;
24
25     ...
26 }
```

Listing 4.8. Generated code for Pattern constraint

```
1 void decode_and_print(uint8_t *buffer, size_t len) {
2     Ieee1609Dot2Data_t *dot2_data = NULL;
3     asn_dec_rval_t rval;
4
5     rval = ber_decode(0, &asn_DEF_Ieee1609Dot2Data, (void
      **)&dot2_data, buffer, len);
6
7     if (rval.code == RC_OK) {
8         printf("Successfully decoded Ieee1609Dot2Data.\n");
9         printf("The constraint check for protocolVersion passed.\n");
10
11        /* Print the decoded structure to stdout */
12        asn_fprint(stdout, &asn_DEF_Ieee1609Dot2Data, dot2_data);
13    } else {
14        fprintf(stderr, "Decode failed. The constraint check may
      have been violated.\n");
15        fprintf(stderr, "Error code: %d, bytes consumed: %zu\n",
      rval.code, rval.consumed);
16    }
17
18    ASN_STRUCT_FREE(asn_DEF_Ieee1609Dot2Data,dot2_data);
19 }
```

Listing 4.9. IEEE 1609.2.1 decoding function

Chapter 5

Conclusion

ETSI employs a wide use of ASN.1 files in its repository, and having an updated open-source software is crucial to studies possible implementation of C-ITS. This thesis project aim was to modify the old *asn1c* to solve the compatibility issues that raised when the new ETSI file was released in 2020. During the problem analysis phase, described in detail in Chapter 3, other missing functionalities were found and decided to be implemented in this code. To sum up, the new added functionalities in this project are

- New functionalities:
  - Generation of constrain checking code for single value constraint applied to any String type.
  - Generation of constrain checking code for pattern constrain.
- Fix to *asn1c* version taken in analysis:
  - Fix on a problem caused by the tag *WITH SUCCESSOR* present in the folder AVM TS 103 882
  - Fix on a problem caused by a complex structure present in the files contained in the folder IEEE 1609.2.1

The project, that was initially intended as a fix of the original software, turned out to include more feature than the original structure and a new layer placed before the existing one, for such reason it was decided to create a brand new branch for this file and calling it *asn2c*. Following the spirit of the main branch, this software will be available in the repository `https://github.com/DriveX-devs/asn2c` for download.
The software is made available to encourage future contributions, starting from the work described in this thesis, more addition can be made to improve the *asn2c* project. The generation of *WITH COMPONENTS* constraint code is missing. In this work the problem is presented and, with the described tools, an implementation can be designed for this constraint and for other constraints that lack implementation. Beside the constraints, this project can be enhanced developing a graphic interface that allow the user to call the program all of its feature in an easier way, also giving a possibility to make a direct

call to the ETSI official repository allowing a easier code generation procedure.
Before closing the thesis, at the moment on which this thesis ending is being written, the owner of the original repository of *asn1c* integrated Microsoft Copilot in its directory. This tool was able, in complete autonomy, to understand the open issue that describe the problem with IEEE 1609.2.1 files, analyze the existing code and generate a working solution. This solution was released after the develop of the one present in this work and takes a complete different path. This event highlight the power of the generative AI and how helpful it can be in the context of code generation, being able to fully manage the cycle of an issue in GitHub. The entire process can be seen in detail in the pull request #268 of the original *asn1c* repository.
In conclusion, the main objective that originated this thesis, together with other issues widely described in this work, has been accomplished creating a working ASN.1 compiler able to correctly work with all the file that implement ETSI ITS-G5 protocol.

Bibliography

[1] Eurostat. Number of cars in the eu reached 253 million in 2021. https://ec.europa.eu/eurostat/web/products-eurostat-news/w/ddn-20230530-1, 2023. Accessed: 2025-09-20.

[2] European Union. Directive 2010/40/eu of the european parliament and of the council of 7 july 2010 on the framework for the deployment of intelligent transport systems in the field of road transport and for interfaces with other modes of transport. https://eur-lex.europa.eu/eli/dir/2010/40/oj/eng, 2010. Accessed: 2025-09-20.

[3] Panayotis Christidis and N.I. Rivas. Measuring road congestion, 03 2012.

[4] Eurostat. Passenger cars by type of motor energy and ownership. https://ec.europa.eu/eurostat/databrowser/bookmark/a66bf785-5fca-4355-8bdf-ea0658c9ff87?lang=en&createdAt=2025-09-20T09:09:13Z, 2025. Accessed: 2025-09-20.

[5] ETSI. Intelligent transport systems (its) committee, 2025. Accessed: 2025-09-20.

[6] ETSI. Etsi en 302 663 v1.2.1 (2013-07): Intelligent transport systems (its); access layer specification for intelligent transport systems operating in the 5 ghz frequency band, 2013. Accessed: 2025-09-20.

[7] ETSI. Etsi en 302 637-2 v1.3.1 (2019-04): Intelligent transport systems (its); vehicular communications; basic set of applications; part 2: Specification of cooperative awareness basic service, 2019. Accessed: 2025-09-20.

[8] ETSI. Etsi en 302 637-3 v1.2.0 (2013-08): Intelligent transport systems (its); vehicular communications; basic set of applications; part 3: Specification of decentralized environmental notification basic service, 2013. Accessed: 2025-09-20.

[9] ITU-T. Recommendation itu-t x.680–x.693. https://www.itu.int/rec/T-REC-X.680-X.693-202102-I/en, February 2021. International Telecommunication Union.

[10] mouse07410. asn1c: Asn.1 to c compiler. https://github.com/mouse07410/asn1c, 2025. Tag: v0.9.27-1366-g9187d2a6, to be compiled.

[11] International Telecommunication Union (ITU). Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1). https://www.itu.int/rec/T-REC-X.208-198811-W/en, 1988. Accessed: 2025-09-05.

[12] International Telecommunication Union. Introduction to ASN.1. https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx, 2025. Accessed: 2025-08-11.

[13] International Telecommunication Union. ASN.1 use cases. https://www.itu.int/

en/ITU-T/asn1/Pages/Application-fields-of-ASN-1.aspx, 2025. Accessed: 2025-08-11.

[14] OSS Nokalva, Inc. About us. https://www.oss.com/company/about-us.html, 2025. Accessed: 2025-09-05.

[15] Objective Systems, Inc. About objective systems. https://obj-sys.com/about/company.php, 2025. Accessed: 2025-09-05.

[16] European Space Agency (ESA). Asn1scc: The asn.1 space certified compiler. https://github.com/esa/asn1scc, 2025. Accessed: 2025-09-05.

[17] Erik Moqvist. asn1tools: Python library for asn.1 parsing and code generation. https://github.com/eerimoq/asn1tools, 2017–2019. Accessed: 2025-09-05, Licensed under MIT.

[18] Pycrate Project. Pycrate: Python library for asn.1, lte, 5g and cryptographic protocols. https://github.com/pycrate-org/pycrate, 2016–2025. Accessed: 2025-09-05.

[19] Lev Walkin and contributors. Asn1c usage guide. https://github.com/mouse07410/asn1c/blob/vlm_master/doc/asn1c-usage.pdf, 2025. Accessed: 2025-09-06.

[20] ETSI. Asn.1 modules for automated vehicle marshalling (avm) — etsi ts 103 882. https://forge.etsi.org/rep/ITS/asn1/avp_ts103882, 2022. Repository folder containing ASN.1 modules of ETSI AVM TS 103 882. Accessed: 2025-09-27.

[21] ETSI ITS Working Group. MSDASN1Module.asn. https://forge.etsi.org/rep/ITS/ECall_HLAP/-/blob/b5c4e7aa3411d92fe60914c55679c3f234e230f5/asn1/MSDASN1Module.asn, 2015. Accessed: 28 September 2025.

[22] OSS Nokalva, Inc. Advanced Constraints in ASN.1. https://www.oss.com/asn1/resources/asn1-made-simple/advanced-constraints.html. Accessed: 2025-10-01.

[23] ITU-T. Recommendation F.515: Multimedia communication service text conversation. https://www.itu.int/rec/T-REC-F.515-200304-I/en, 2003. Accessed: 2025-10-01.

[24] ETSI ITS Working Group. CDD TS 102 894-2 ASN.1 Module Repository. https://forge.etsi.org/rep/ITS/asn1/cdd_ts102894_2. Accessed: 2025-10-01.

[25] Vanetza contributors. Pull Request #223: Add support for extended types in ASN.1 compiler. https://github.com/riebl/vanetza/pull/223, 2020. Accessed: 2025-10-02.

[26] The Open Group. Regular Expression Definitions — <regex.h> (POSIX Base Specifications, Issue 7). https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/regex.h.html, 2018. Accessed: October 2025.

[27] International Organization for Standardization. ISO/IEC 9945:2003 — Information technology — Portable Operating System Interface (POSIX®). https://www.iso.org/standard/17841.html, 2003. Accessed: October 2025.

[28] Riccardo Riebl and contributors. Vanetza: An open-source implementation of the ETSI ITS-G5 protocol stack. https://github.com/riebl/vanetza, 2015. Accessed: October 2025.