An ASN.1 compiler[1] for embedded/space systems

George Mamais[i], Thanassis Tsiodras[i], David Lesens[ii], Maxime Perrotin[iii]

[i] Semantix Information Technologies, K.Tsaldari 62, Poligono 114 76, Athens, Greece
{gmamais, ttsiodras}@semantix.gr
[ii] Astrium Space Transportation. Route de Verneuil, BP 3002, F-78 133 Les Mureaux Cedex, France
David.Lesens@astrium.eads.net
[iii] ESTEC. Keplerlaan 1. PO Box 299, NL-2200 AG Noordwijk, The Netherlands
Maxime.Perrotin@esa.int

This paper presents ASN1SCC, an open source[2] ASN.1 compiler that generates C/C++ and SPARK/Ada code suitable for low resource environments such as space systems. Moreover, the compiler can produce a test harness that provides full statement coverage in the generated code, and therefore significantly improves its quality. This paper also presents ACN, a new ASN.1 encoding that allows protocol designers to completely control the format of the encoded ASN.1 stream and hence integrate ASN.1 applications with legacy ones. With ASN.1 and ACN, various space protocols such as PUS[3] can be modeled and with the usage of this ASN.1 compiler get automatic implementations of the encoders and decoders. Finally, the ASN.1 compiler can translate an ASN.1/ACN definition into an Interface Control Document (ICD), thus allowing interoperability with projects and people who don't know/use ASN.1.

1. Introduction

ASN.1 is a joint ISO/IEC and ITU-T standard that defines a flexible notation for describing data structures. It represents data structures in an abstract, but formal manner - meaning that the descriptions are platform and language agnostic, and that most importantly, they are validated by a machine (a program). An ASN.1 compiler is a computer program that translates an ASN.1 specification into code of a target programming language, such as C or Java. The generated code consists of two parts: (a) a set of data structure declarations (types) which are semantically equivalent with those defined in ASN.1 and (b) a set of functions, known as encoders and decoders, which convert instances of the data structure declarations into a stream of bytes - and vice versa. The rules of the encoding and decoding process are defined by one of the standardized ASN.1 encoding schemes (such as BER, uPER, XER, etc) and are completely independent of the chosen programming language, processor architecture or operating system. Without ASN.1, the communication between an Ada process running inside an RTEMS/SPARC/LEON environment and a C process running on Windows/x86, requires manual implementation of message encoders and decoders in both processes. This kind of manual coding is tedious and error prone, since developers must deal with issues like integer word size, endianness, structure alignment, etc. If ASN.1 is utilized, the developers don't worry about any of the above issues since the encoders and decoders of the exchanged messages are automatically generated by the ASN.1 compiler.

The above features make ASN.1 an ideal choice for inter-process communication, especially in low resource environments. However, the existing COTS (commercial, off-the-shelf) ASN.1 compilers have some significant issues when targeting low resource platforms. In particular:

(a) The generated encoders and decoders as well as the run-time libraries use dynamic memory functions (e.g. `malloc`-ed, heap data). In embedded environments, usage of dynamic memory is sometimes prohibited (e.g. satellites in space).
(b) Programming languages important for embedded platforms are not supported (e.g. Ada).
(c) Although existing ASN.1 compilers support many ASN.1 encodings, they provide no easy mechanism[4] for the protocol designer to affect the binary format of the encoded messages. Interworking with existing, non-ASN.1 based systems, is therefore very difficult.

[1] The ASN.1 compiler presented in this paper has been funded by ESA/ESTEC, contracts 21111/07/NL/FM/na and 22259/09/NL/CBI
[2] http://www.semantix.gr/asn1scc/
[3] Packet Utilization Standard, ECSS-E-70-41
[4] ECN (Encoding Control Notation) is extremely complex and is currently supported by only one vendor.

2. ASN1SCC

ASN1SCC (http://www.semantix.gr/asn1scc) is an open source ASN.1 compiler that was built to meet the requirements of embedded/space platforms. It supports the major ASN.1 encodings: (a) Unaligned Packed Encoding Rules - uPER (b) Basic Encoding Rules - BER (c) XML Encoding Rules - XER and (d) ASN.1 Control Notation which allows the protocol designer to contol the binary stream (see section iv). As opposed to other ASN.1 tools, both the compiler and its run-time library are open-source. Moreover, it has a set of unique features that distinguish it from other ASN.1 compilers, and are briefly described in the following paragraphs.

i) No dynamic memory

The code generated by ASN1SCC, as well as the run-time library, never use dynamic memory functions (such as `malloc`). All memory requirements (i.e. the size of the encoded and decoded buffers for each ASN.1 message) are calculated at compile time. By doing so, all required memory can be statically reserved at compile-time, thus guaranteeing that there will be no failure due to lack of memory at run-time.

For example, an ASN.1 type describing an array of integers...

```
EXAMPLE-MODULE DEFINITIONS AUTOMATIC TAGS ::= BEGIN
    AnArray ::= SEQUENCE (SIZE(1..10)) OF INTEGER
END
```

...triggers the generation of the following macro definition and C structure from ASN1SCC:

```
#define AnArray_REQUIRED_BYTES_FOR_ENCODING 91

typedef struct {
    long nCount;
    asn1SccSint arr[10];
} AnArray;
```

Notice that the maximum size of the encoded data is available at compile-time, via the macro *AnArray_REQUIRED_BYTES_FOR_ENCODING*. This means that the user code can statically reserve the necessary space at compile-time, in global or static variables, thus guaranteeing the availability of the necessary space:

```
/* As global */
char encodedStream[AnArray_REQUIRED_BYTES_FOR_ENCODING];

/* As static data in a function */
void foo(...)
{
    static char encodedStream[AnArray_REQUIRED_BYTES_FOR_ENCODING];
    ....
}
```

The variable length arrays of ASN.1 (i.e. *SEQUENCE SIZE*(S_{min} .. S_{max}) *OF T* are mapped by ASN1SCC to C structures. These structures contain an inline fixed-size array of T with size S_{max} (i.e. the maximum possible extent of the array) as well as an integer field that stores the actual number of the elements used.

The fact that the actual data are stored as an inline array and not as a heap-allocated block pointed to by a pointer (or similarly, a linked list with pointers to heap allocated objects), means that *sizeof(AnArray)* will always represent the maximum memory needs for the target type. This is true regardless of the complexity of the type (e.g. arrays of sequences containing arrays, etc) - and allows reservation of all the necessary memory space at compile-time.

ii) Automatic Statement Coverage

Critical software like space applications, must meet a set of guidelines - such as specific coding conventions, or thresholds about branch and statement coverage levels. ASN1SCC itself has its own regression checking suite, where thousands of ASN.1 grammars are used to drive the following sequence (for each ASN.1 grammar):

- ASN1SCC processes the grammar and generates encoders and decoders for its types.
- A template "main" function is created that encodes the main (root) grammar message, and then decodes it from the encoded data.
- The generated "main" combined with the ASN1SCC generated encoders and decoders is compiled, and the generated binary is executed.
- Checks are performed to verify that the message field data remained the same across encoding and decoding.
- Further checks are done to see that no internal abnormal state was encountered in the encoder and the decoder during this “round trip” (even if this internal abnormality did not result in any externally visible errors). This includes memory access validations from Valgrind[5].

This test suite provides a baseline of confidence for the correctness of ASN1SCC and the code it generates. However, a valid question posed by a number of early users of ASN1SCC (space companies and ESA) was...

"How can we have increased confidence that code generated by ASN1SCC will correctly process all the - theoretically infinite! - messages that are possible under a particular ASN.1 grammar?"

The caveat here being that, obviously, a project-specific grammar cannot be tested as part of the ASN1SCC quality assurance process - since ASN1SCC is not conceived with any particular grammar in mind.

To address this issue ASN1SCC was enhanced further: it can now automatically generate a set of unit tests for a given input grammar. These unit tests are in fact sets of ASN.1 variable assignments - i.e. data assignments for the grammar's types. The key point is that these data assignments, upon encoding and decoding, exercise the ASN1SCC-generated encoders and decoders to 100% statement coverage. At the end of this test, the user knows for a fact, that (a) the encoders and decoders were exercised by the automatically generated unit tests so that all their code was executed, without any abnormal state arising at runtime, and (b) that the message data were perfectly preserved in the round trip (original data => encoded message => decoded message => same data).

Assume for example that a simple ASN.1 grammar describes messages containing double precision numbers:

```
EXAMPLE-MODULE DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
    MyTestMessage ::= REAL
END
```

In the course of encoding, decoding and otherwise handling these messages, the code generated by ASN1SCC has to consider all the possible "states" for this number:

```
if (value == 0.0)
    // auto-generated code for handling zero
else if (value == +∞)
    // auto-generated code for handling positive infinity
else if (value == -∞)
    // auto-generated code for handling negative infinity
else
    // auto-generated code for handling "normal" case
```

[5] *Valgrind: an instrumentation framework for dynamic analysis of executables* (http://valgrind.org/)

When fed with this grammar, ASN1SCC generates a number of test cases, declared as ASN.1 variable assignments:

```
EXAMPLE-MODULE-TESTS DEFINITIONS AUTOMATIC TAGS::=
BEGIN
    IMPORTS MyTestMessage FROM EXAMPLE-MODULE;

    test1 MyTestMessage ::= 0.0            -- zero
    test2 MyTestMessage ::= PLUS-INFINITY  -- positive infinity
    test3 MyTestMessage ::= MINUS-INFINITY -- negative infinity
    test4 MyTestMessage ::= 3.14           -- normal number
END
```

ASN1SCC compiles these test cases and generates code for the target languages. The generated code, in tandem with a specially made test harness, is subsequently compiled by the compiler - and the resulting executable is then spawned under a coverage checking tool[6]. For each individual message ("test1", "test2", etc) the message content is encoded, decoded and verified to survive the round trip. At the end of the execution, the coverage checker reports the statement coverage in the encoders and decoders, and the test harness verifies that this is 100%.

Notice that the process is automated - there's no human involvement. If, for example, the generated test messages do not exercise a part of the code, the user will then get a notice from the coverage checker, that a part of the encoder/decoder is not tested. In that case, he can indicate this to the ASN1SCC developers (so that the required additional test case is generated) - or he can still identify the parts in question, and manually create additional tests for these cases, to reach 100% coverage. This feature is not currently available in any other ASN.1 compiler.

iii) SPARK/Ada support

To increase the quality of the generated sources in environments where very high reliability is expected, the Ada versions of the encoders/decoders generated by ASN1SCC use SPARK/Ada annotations.

The SPARK language consists of a restricted, well-defined subset of the Ada language that uses annotated meta-information - in the form of Ada comments. This meta-information describes the desired component behavior and the individual runtime requirements. It therefore allows static analysis to be performed at compile-time as a further, automated check on program correctness. The static analysis verifies code invariants, preconditions and postconditions – that is, it applies Design by Contract principles to accurately formalize and validate the expected runtime behavior of the ASN1SCC-generated encoders and decoders.

Assume for example the following Ada function definition, which divides two integers:

```
FUNCTION DIV (dividend: INTEGER; divisor: INTEGER) RETURN INTEGER;
```

The programmer implicitly knows that the divisor must never be zero. However, this knowledge is available only to the programmer, and not to the compiler. Therefore if this function is called by user code that calculates the arguments at run-time via an algorithm, the code may end up being called with a divisor value of zero - and an error will then appear and potentially crash the application at run time. To address this, SPARK allows programmers to enrich their interfaces with pre and post conditions. For example, the above definition in SPARK can be annotated as follows:

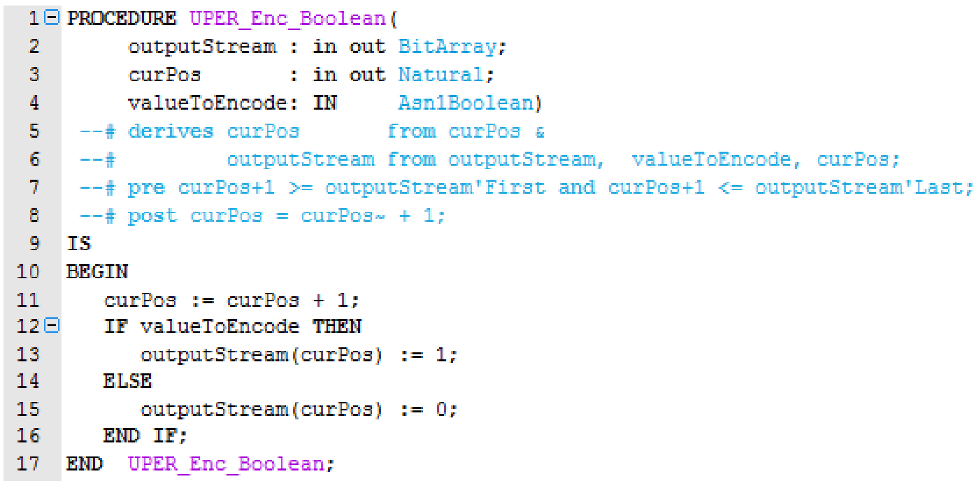
```
FUNCTION DIV (dividend: INTEGER; divisor: INTEGER) RETURN INTEGER;
--# pre divisor <> 0
```

The line starting with --# is a SPARK annotation which explicitly informs the SPARK examiner that the DIV function *requires* the divisor parameter to never be zero. This means that the SPARK examiner must check *at compile time* all the places where the DIV function is called and make sure that it is *never* called with divisor equal to zero. If this is not the case, the examiner reports an error.

[6] Gcov, coverage checking (http://gcc.gnu.org/onlinedocs/gcc/Gcov.html)

ASN1SCC's SPARK backend emits encoders and decoders that are annotated with SPARK annotations, thus allowing the static analysis to detect invalid usage (data-wise) of the API. In fact, PER-visible ASN.1 constraints are transformed into semantically equivalent SPARK annotations, and will therefore be thoroughly checked by static analysis, guaranteeing that a buffer overflow during encoding/decoding is impossible.

For example, the following procedure is used in the encoding of a Boolean type in uPER. If the encoded value is true, then bit value 1 is written in the uPER stream - otherwise bit value 0 is written.



```
 1  PROCEDURE UPER_Enc_Boolean(
 2       outputStream : in out BitArray;
 3       curPos       : in out Natural;
 4       valueToEncode: IN     Asn1Boolean)
 5   --# derives curPos        from curPos &
 6   --#         outputStream from outputStream,  valueToEncode, curPos;
 7   --# pre curPos+1 >= outputStream'First and curPos+1 <= outputStream'Last;
 8   --# post curPos = curPos~ + 1;
 9  IS
10  BEGIN
11     curPos := curPos + 1;
12     IF valueToEncode THEN
13        outputStream(curPos) := 1;
14     ELSE
15        outputStream(curPos) := 0;
16     END IF;
17  END  UPER_Enc_Boolean;
```

The *derives* annotation informs SPARK about the dependencies of outputs from the input values. The *pre* annotation says that whenever this function is called the value of *curPos* counter plus one must be within the bounds of the outputStream array. Finally, the *post* annotation says the value of the *curPos* counter will be increased by one by the end of this function.

The important impact of these annotations is that SPARK will be able to enforce these restrictions at compile time and thus an out of range exception in lines 13 and 15 is impossible.

iv) Integration with legacy systems – the ACN encoding

The major benefit of ASN.1 is that the encoding and decoding process is independent of the programming language, the hardware platform and the Operating System. Moreover, the standardized ASN.1 encoding schemes offer additional and significant benefits, such as speed and compactness for PER or decoding robustness for BER, etc. However, the existing ASN.1 encodings provide no means for the protocol designer to control the final encoding (i.e the binary format at the bit level). This is a problem for situations where a new ASN.1-based system has to communicate over a binary protocol with an existing legacy system. For example, the PUS (Packet Utilization Standard), which is used in space missions to encode Telemetry/Telecommands, cannot be specified in ASN.1 using any of the existing encodings (BER,PER,etc).

To address this issue, we designed and implemented a new ASN.1 encoding, known as ACN (ASN.1 Control Notation). ACN allows protocol designers to control the format of the encoded messages at the bit level. In ACN, users can specify how each ASN.1 type will be encoded. Attributes can be set, such as the bit length of an integer, its endianness (big/little), its alignment etc. Moreover, for aggregate fields such as SEQUENCE, CHOICE and SEQUENCE OF the user can define optionality patterns, choice determinants, length fields etc.

For example, to encode an unsigned integer in16 bits and align it to the next octet start (i.e. a stream offset in bits which is a multiple of 8), you would...

ASN.1	MyInt1::=INTEGER (0..2000)
ACN	MyInt1[encoding pos-int, size 16, align-to-next byte]

In the above example, the protocol designer specified via ACN that the ASN.1 type *MyInt1* will be encoded as a positive integer, using 16 bits, and will always be byte aligned in the encoded buffer.

A more advanced example of ACN, based on PUS:

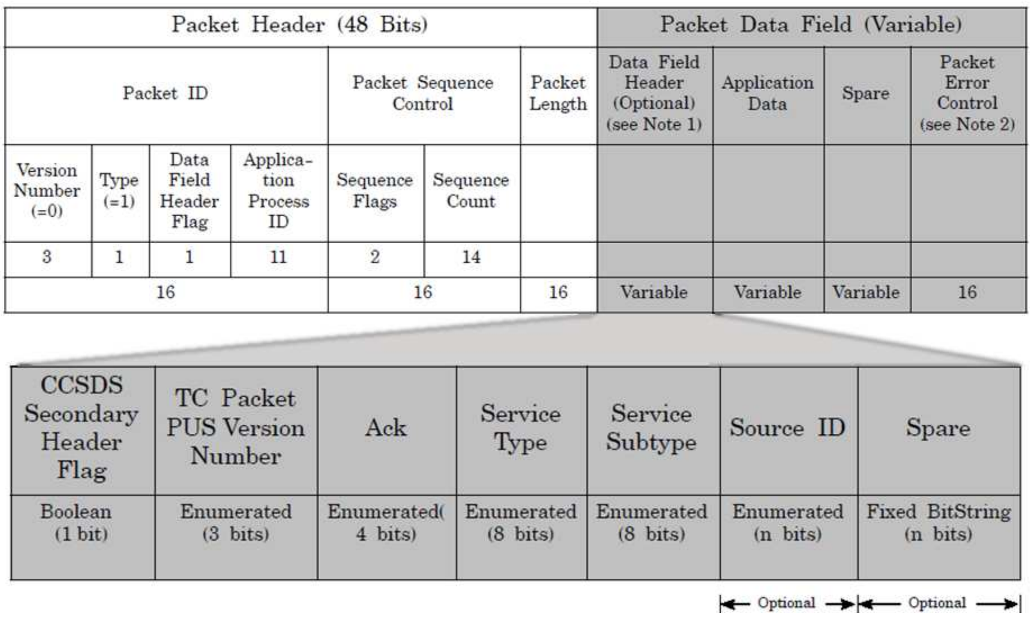


Figure 1: *Automatic PUS encoding via ACN*

Figure 1 shows a PUS packet. It consists of two main parts: (a) the Packet Header (area with the white background) and (b) the Packet Data Field (top right grey area).

The Packet Data Field part is actually a composite field consisting of multiple sub-fields. The Data Field Header has two enumerated fields (service type and service subtype) which determine the actual form of the “Application Data” field.

In other words, the “Application Data” field is a CHOICE type where the active alternative is determined by the combination of Data Field Header fields *service type* and *service subtype*. In ASN.1 this can be modeled as follows:

```
PacketDataField ::= SEQUENCE {
    ...
    dataFieldHeader      DataFieldHeader,
    msg                  ApplicationData,
    ...
}

DataFieldHeader ::= SEQUENCE {
    ...
    service-type     INTEGER (0..255),
    service-subtype  INTEGER (0..255),
    ...
}

-- example of application data
-- (based on service-type/subtype combination)
ApplicationData::= CHOICE {
    a1      MessageA, -- when the combination is 100,120
    a2      MessageB, -- when the combination is 101,120
    a3      MessageC, -- when the combination is 101,121
    ...
}
```

With the appropriate ACN syntax, one can bind the elements of this ASN.1 grammar into a valid PUS specification:

```
PacketDataField {
     ...
     dataFieldHeader [],
     msg < dataFieldHeader.service-type, dataFieldHeader.service-subtype >[],
     ...
}

DataFieldHeader {
     ...
     service-type     [encoding pos-int, size 8],
     service-subtype  [encoding pos-int, size 8],
     ...
}

ApplicationData <INTEGER:servID, INTEGER:subServID> [] {
     a1     [present-when servID==100 subServID==120],
     a2     [present-when servID==101 subServID==120],
     a3     [present-when servID==101 subServID==121],
     ...
}
```

The ACN *present-when* keyword is used to match the specific combinations of values with the type that the *ApplicationData* CHOICE is carrying. Notice that:

1. The *ApplicationData* has a parameterized ACN encoding. This is shown with the angle brackets ('<',' >'). Parameterized encoding means that this type cannot be decoded independently - it needs two extra values. In other words, the ACN decoder function for the *ApplicationData* type will have two extra parameters that must be passed by the caller.

2. Likewise, the msg field in the *PacketDataField*, which is a reference type to the parameterized type *ApplicationData*, has two additional arguments, the *dataFieldHeader.service-type* and *dataFieldHeader.service-subtype*. So, a two way binding has been established between the two fields in the *DataFieldHeader* and the CHOICE type *ApplicationData*. Two way binding means:

   - during the decoding process, the CHOICE *ApplicationData* will read the values of service-type and service-subtype in order to be decoded correctly
   - during the encoding process, the values of *service-type* and *service-subtype* will be updated automatically based on which of the alternatives a1, a2, a3 is present in the CHOICE *ApplicationData*.

3. The *present-when* syntax within the CHOICE *ApplicationData* expects a boolean expression (in fact, a boolean AND expression), i.e. when all comparisons match, the selected CHOICE target is used.

v) Automatic ICDs

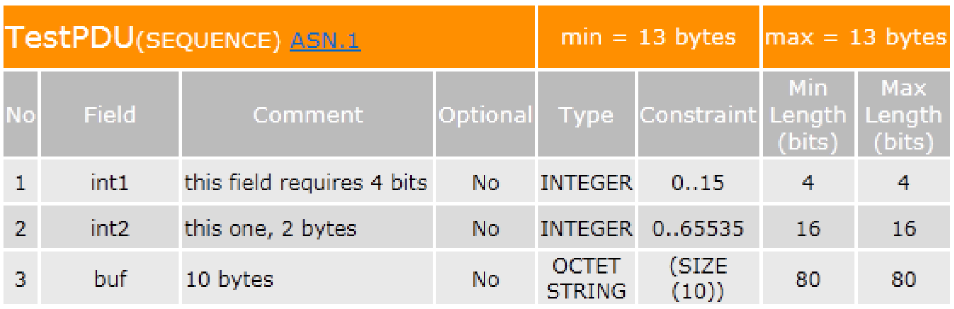
Interface Control Documents (ICDs) describe the binary format of messages exchanged between entities (the "wire format"), and are widely used in the space domain e.g. in the specification of space/ground interfaces.

ASN1SCC can automatically create ICDs for a given ASN.1 grammar. This allows users who are not familiar with ASN.1 to easily understand the structure of the encoded messages, and if they so wish, manually implement the required encoding in their target environment, and interoperate. The ICD generator supports the uPER and ACN encodings.

For example in the following ASN.1 message…

```
TestPDU ::= SEQUENCE {
      int1   INTEGER(0..15), -- this field requires 4 bits
      int2   INTEGER(0..65535), -- this one, 2 bytes
      buf    OCTET STRING (SIZE(10)) -- 10 bytes
}
```

... the compiler will create an Interface Control Document that will contain the following table:



TestPDU(SEQUENCE) ASN.1				min = 13 bytes		max = 13 bytes	
No	Field	Comment	Optional	Type	Constraint	Min Length (bits)	Max Length (bits)
1	int1	this field requires 4 bits	No	INTEGER	0..15	4	4
2	int2	this one, 2 bytes	No	INTEGER	0..65535	16	16
3	buf	10 bytes	No	OCTET STRING	(SIZE (10))	80	80

Figure 2: *ICD generator output*

The above is an example of the visual layout of the generated document which demonstrates why this tabular / visual way of defining the data structures is more comprehensible for people not familiar with ASN.1.

3. Use of ASN.1 in space application

The interest of the ASN.1 data modelling language and of the associated tools (code and ICD generators) has been assessed by Astrium by retro-engineering a part of the ATV (Automated Transfer Vehicle) program.

Figure 3: *ATV Jules Verne, during its rendezvous with the ISS, on the 3rd of April 2008*

When code and documentation (ICD) generation techniques from ASN.1 are used, it can decrease dramatically the number of potential inconsistence:

a. Between the software and its documentation: The definition of data has to be performed only once (in the data modelling language). The syntax and semantics checker ensure the quality of this description. Code and ICD are naturally consistent (modulo the quality and certification of the generation tools). Furthermore, whenever a change is made in the original model, it becomes immediately visible to both ICD and code, keeping them synchronised.

b. Between communicating software components: When communicating software components are developed in different programming languages (Ada, SCADE, C…) or are running on different types of processors (ERC32, Leon…), they may use different incompatible memory representation of data. This requires generally a manual implementation of translation algorithms, which require a perfect knowledge of the compilers and of the hardware architecture. The ASN.1 toolset decreases dramatically the cost of the software development and suppresses the risks of incorrect implementation of the translation code.

```
BLOCK3-HEADER ::= SEQUENCE   {
    true-dtg-axis-to-atv-msu1 SEQUENCE ( SIZE(6) ) OF REAL ,
    true-dtg-axis-to-atv-msu2 REAL ,
    true-ssu-axis-to-atv REAL,
    dtg-consistency-lines-msu1-dtg1-y REAL,
    dtg-consistency-lines-msu1-dtg3-x REAL,
    dtg-consistency-lines-msu1-dtg3-y REAL
}
```

Figure 4: *Avionics / Software interface description*

c. For the Telemetry/Telecommand function: The TM/TC function implements the interfaces between the ground control center and the spacecraft. Often developed by two distinct industrials, using generally different programming languages and hardware architectures, formalizing the shared data is often a challenge. ASN.1 offers a large number of features allowing the full description of this interface, as specified by the ESA standard “Ground systems and operations - Telemetry and telecommand packet utilization” (ECSS-E-70-41).

```
USER-DATA-TC ::= CHOICE   {
– Distribute Trouble Shooting Commands
   tc-2-1 SEQUENCE   {
   – A unique identification of the bus where the Data is to be
   – written
      bus-id BUS-ID(ALL EXCEPT nop),
      – MIL-1553B Message Command word
      command-word COMMAND-WORD,
      – Identifies the MIL-STD-1553B command type
      command CHOICE   {
         transmit-command-data NULL,
         – MIL-1553B Command Data. This field is present if the Command
         – Type is 'Troubleshooting TC'.
         receive-command-data BUS-DATA
      }
   } ,
   – Distribute Low-Level Commands
   tc-2-2 SEQUENCE   {
   – Data corresponding to the LLC to be uploaded
      llc LLC
   } ,
```

Figure 5: *Ground / Board interface (TM/TC) description*

ASN.1 is today envisaged for the future generation of European launcher. The digital architecture of this launcher will be:

a. Distributed: The flight software will be dispatched on several processors,
b. Partitioned: Two pieces of software will run on a single board with a Time and Space Partitioning (TSP) hypervisor and
c. Heterogeneous: The software will be partially coded in Ada, partially in C, partially automatically generated from SCADE

ASN.1 is therefore the main candidate to capture the interfaces between the processors, between the partitions and between the heterogeneous pieces of code.

The Ada/SPARK version of the ASN1SCC compiler could especially be used to automate the tedious and error-prone coding of the data encoding.

Figure 6: *Future launcher*

4. Future work

In the space domain, there are families of protocols that are customized on a mission-specific basis. PUS, for example, has a fixed mission-agnostic part that is always the same, and a mission-specific one. We are currently introducing template support in ASN1SCC which will allow the users to “partially” define message structures that will be instantiated on a mission-specific basis when the mission-specific parts are known. This will allow optimal re-use of message specifications across missions. In this way it will be possible to aggregate ASN.1 specifications in a library of reusable components.

For example, assume that there is a family of missions that use messages of the following type:

mission A:	`Frequencies::= SEQUENCE (SIZE(10)) OF INTEGER`
mission B:	`Frequencies::= SEQUENCE (SIZE(25)) OF REAL`
mission C:	`Frequencies::= SEQUENCE (SIZE(1000)) OF INTEGER`

Both the size of the frequencies array as well as the type used to represent frequencies is configured in a mission-specific basis. "Reuse" in such a context consists in copying and modifying the previously used definition (by way of example). Instead, ASN.1 templates would allow a generic template to be specified once in a simple and concrete manner. E.g. the reusable template component would be:

```
PROBA-MISSION-TEMPLATES DEFINITIONS AUTOMATIC TAGS::= BEGIN

FrequenciesTemplate{INTEGER:someLength, SomeType } ::=
SEQUENCE (SIZE(someLength)) OF SomeType

END
```

while the mission specific would be:

```
PROBA3-MISSION DEFINITIONS AUTOMATIC TAGS::= BEGIN
FROM PROBA-MISSION-TEMPLATES IMPORT FrequenciesTemplate;

Frequencies ::= FrequenciesTemplate{100, REAL}

END
```

Note that from an engineering perspective this is not about the number of keystrokes involved or the inconvenience of having to copy-paste and modify the definition used in the last mission. Without ASN.1 template support, the "generic" specification of what a frequencies table should look like is given in a human language – where as with ASN.1 template support, the specification is given in a formal notation, can be put under source control and can be leveraged by all kinds of tools (not just the ASN1SCC).

5. Conclusion

We presented an open-source ASN.1 compiler which was specifically made to target embedded platforms and their needs. The compiler includes support for features that significantly improve the correctness and verifiability of the generated code (like SPARK and automatic test case generation), and also supports ACN, which allows the user to control the binary format of the encoded messages. Moreover, it automatically generates interface control documents (ICDs) thus allowing seamless interoperability with non-ASN.1 entities. As demonstrated by the preliminary assessment from Astrium, ASN1SCC can play an important role in the implementation of safety critical applications for embedded platforms.