Applicable objectives: O-1, O-2

Toolchain to connect EDS and TASTE

Michał Kurowski[1], Filip Demski[1], Jakub Rachucki[1], Maxime Perrotin[2], David Perillo[2]

1) N7 Space sp. z o.o. 2) European Space Agency

Abstract

The paper presents some of the achievements and lessons learned from creating a translator between SOIS EDS and TASTE models, as well as a demonstration use case which involved creation and evaluation of simple EDS sensor and actuator specifications. The prepared EDS models, controlling a combined LIDAR sensor, were translated into ASN.1, Interface View components as well as SDL processes, and then deployed on a real hardware platform – ARM Cortex-M7 SAMV71. Practical feedback relevant to the authoring process, CCSD 876-0-B-1 standard and TASTE is provided.

Introduction

TASTE[1] is ESA's MBSE toolchain targeting the development of heterogeneous systems, which acts as a hub for different languages/technologies (C, Ada, SDL, ASN.1, Simulink, etc.) and provides code generation capabilities. CCSDS 876-0-B-1 Spacecraft Onboard Interface Services Electronic Data Sheets[2] (EDS) is an XML specification for capturing data, interface and behaviour models. As there is a significant – though not complete – overlap between TASTE and EDS model scopes, there is a possibility for model-to-model translation, which could:

- provide indirect authoring tools for EDS – via translation from e.g., ASN.1, which has proper tooling,
- provide code generation for EDS – via translation to e.g., SDL, which has a code generator,
- strengthen TASTE function as a technology hub, providing interoperability with EDS.

In order to achieve these goals, a translator was created and integrated with SpaceCreator, which serves as the Integrated Development Environment for TASTE. This decision allowed to re-use significant amounts of ASN.1/ACN and Interface View related code, as well as create a generic model translation framework which is already used in another project. The following transformations are supported:

- from EDS data model to ASN.1/ACN – which can be further translated into C or Ada for data structure definitions, as well as packet encoders and decoders,
- from EDS interface model to TASTE Interface View – which, after adding connections between the component interfaces, acts as a logical architecture, from which "glue-code" can be generated,
- from EDS behaviour model to SDL – which can be further translated into C or Ada implementations of state machines, used for application logic and data handling,
- from ASN.1/ACN to EDS data model,
- from Interface View to EDS interface model.

Translation from SDL to EDS behaviour model (state machines) was considered out of scope of this study due to the complexity of the task, and the fact that only a subset of SDL could be supported by EDS in a feasible manner. Some of the reasons for this assessment are mentioned further in this paper.

SOIS EDS is intended to replace traditional interface control documents and proprietary data sheets with machine readable interface specifications. The format is suitable for describing devices and applications in terms of their data interfaces and internal behaviour. One of the goals of the project was to evaluate the EDS capabilities in "real-life" applications, with focus on describing sensors and actuators. To this end, the code developed in the context of this activity has been validated using a purposely developed embedded

system. In order to provide a host hardware platform, a TASTE runtime was developed for ARM Cortex-M7 SAMV71 microcontroller. It is based on FreeRTOS[5] and designed to be as lightweight as possible. For easier interoperability and taking up an opportunity to provide a streamlined alternative to PolyORB (the default middleware used in TASTE), another TASTE Linux runtime was also provided.

One of the identified limitations common to EDS Component Implementation and the SDL dialect they are translated to is the unavailability of native constructs for "pointers", memory operations, nor the concept of interrupts. As a consequence, our approach uses a proxy component – HardWare Access EDS (HWAS) – exposing interfaces described in EDS (for access by other EDS components), with an implementation in C (enabling access to memory and interrupts). Its implementation is specific to ARM SAMV71 microcontroller. However, the interface declarations could be reused for various other platforms.

Demonstration Use Case

The goal of the study was to create a complete demonstration of an MBSE development for a small platform using a combination of TASTE and EDS components. A similar goal in a preceding activity [4] was realized through designing a Cube Sat class system. For diversification, in this activity a LIDAR instrument was proposed. The system is enclosed in a 3D printed case and contains:

- LEDs for visual output, driven via GPIOs (an actuator),
- mock "Sun sensor", based on a greyscale sensor connected to SAMV71's ADC (a sensor),
- LIDAR itself, which combines UART communication with COTS TF Luna sensor, contact sensors read via GPIO and a stepper motor driven via GPIO (a combined set of sensors and an actuator).

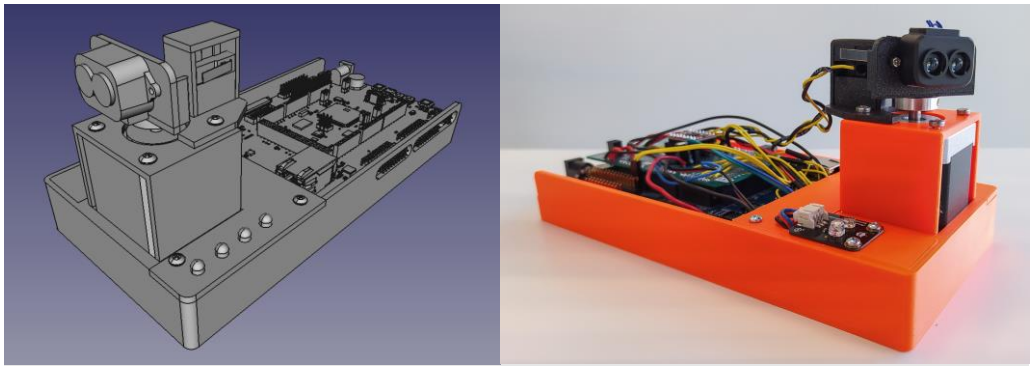

Figure 1 LIDAR instrument: 3D-printed housing design (left), assembled module (right)

The following EDS specifications were created:

- HWAS – containing only the data types, as well as memory and interface declarations,
- PIO HWAS – for manipulating SAMV71's GPIO, accessing the HW via HWAS,
- UART HWAS – for handling SAMV71's UART, accessing the HW via HWAS and PIO HWAS,
- AFEC HWAS – for reading analogue signals using SAMV71's ADC, accessing the HW via HWAS,
- SunSensor – for acquiring readings from the greyscale sensor, accessed via AFEC HWAS,
- TfLuna – for acquiring range data from TF Luna LIDAR, accessed via UART HWAS,
- MP6500 – for controlling the stepper motor via a HW motor driver, accessed via PIO HWAS,
- Lidar – forming the combined sensor/actuator, forwarding the data from TfLuna, and managing its rotation by controlling the stepper motor via MP6500 and contact sensors via PIO HWAS.

All EDS specifications were divided into two parts – one with declarations (EDS "Packages" for re-use via XInclude by other EDS) and actual component instantiations (Components in EDS "Devices").

Results and Lessons Learned

All the EDS specifications for the hardware peripherals – physical (PIO, AFEC, UART) and virtual (LIDAR, MP6500, SunSensor) – were successfully implemented, deployed onto the target hardware and tested, confirming the feasibility of fully describing onboard hardware devices via Electronic Data Sheets.

The design of the translator involved discussions between N7 Space and ESA regarding the exact interpretation of certain EDS constructs and their practical mapping onto TASTE. For instance, EDS "sync" primitives were mapped onto TASTE "protected" interfaces, which are internally translated into synchronous function calls. This was deemed as the best analogy, contrasting them to "async" primitives mapped onto TASTE "sporadic" interfaces, internally translated into asynchronous message queues. This however caused issue with the interpretation of the "sync" "allArgTypes" command pattern (see Figure 4-3 of [2]), which contains a synchronous looped update. As a result, this particular pattern is not supported. Similarly, considerations regarding EDS AlternateSet lead to its mapping onto ASN.1 "choice" construct. However, compared to the ASN.1 "choice" construct, the EDS AlternateSet does not unambiguously define a construct for selecting the specific Alternative. Our approach was thus to select the specific alternative in the AlternateSet based on an algorithmically selected fixed value field. While EDS enables detailed encoding specification for all explicit command arguments, the implicit string "transaction" attribute is not associated with any encoding and therefore it was assumed that its interpretation is implementation dependent. As a consequence of the above, EDS implementations not sharing the same assumptions and limitations may exhibit compatibility issues.

Once the translator was implemented, the required EDS specifications were created by an embedded engineer with no previous exposure to MBSE, TASTE and EDS, providing a good perspective from a validation point of view. While ASN.1 and Interface View transformation to EDS was available, defining the data and interfaces directly in EDS XML without any intermediate translation steps was deemed more straightforward given the simplicity of the needed constructs, and was executed without major issues. The most problematic concept was the understanding of EDS interface directions (provided/required), as the translation into TASTE models caused some interfaces to be split into pairs (separate for input and output/notify) or change directions, depending on whether the given command is sync or async. For example, EDS provided interface with async indication was transformed into TASTE required interface with input argument. This issue was however quickly resolved after several concrete examples were provided and the system was demonstrated in action. On the other hand, definition of EDS component implementations has proven itself to be difficult and time consuming. The translation from SDL to EDS was not available, and N7 Space was not in possession of any dedicated tools, except for a schema aware XML editor with limited autocompletion and validation features. Understanding of complex state machines, with long lists of actions (especially when embedded in conditional statements), was difficult, making it a challenge to both write and review them. Additionally, what was in the end executed on the target hardware – and so possible to inspect via a debugger – was not the EDS state machine itself, but the assembly of Ada code generated from SDL, which was derived from the input state machine. This made troubleshooting quite challenging. The issue was mitigated to some extent by first writing the needed code in C (using the same API as the one available to EDS), testing it, debugging, and then translating it manually to XML. While the approach has proven successful and made certain things easier, it was time consuming. One of the first issues reported by the embedded engineer was the lack of authoring tools, and the need to directly edit the format that is supposed to be just machine readable. Another issue is that while EDS commands have both input and output/notify arguments, activities used within state machines have arguments without any mode, and thus understood as being input only. This makes writing helper utilities a bit more difficult, as any calculation results must be stored in state machine variables. SDL allows to define state machines in a more compact, and flexible way than EDS, which was the

primary cause of the initial assessment of SDL to EDS translation feasibility. For example, each EDS transition has “fromState” and “toState” attributes, explicitly defining the single start and end states by name. SDL on the other hand contains special symbols (*, -) and state lists, which make writing sets of logically identical transitions much faster, more readable and easier to maintain. In EDS, for each transition trigger and start state, the single end state is explicitly defined. EDS provides a way to enable or disable the given transition at runtime using transition guards. On the other hand, SDL allows to perform computation during the transition and decide on the target state, as well the state change itself in runtime. In order to emulate this crucial feature, embedded engineer decided to write “internal” state machines manually, using component variables for state storage and generic operations available for activities to manage the transitions. It would be beneficial for EDS to define the target transition state as the result of activity processing, for example by allowing to omit the “to” attribute of the transition, and introducing an Activity Body element indicating the target state. EDS “on timer” transition has a fixed “nanosecondsAfterEntry” attribute, in contrast to a schedulable SDL timer. In addition, SDL contains the concept of continuous signals, absent from EDS. While it could be emulated via high-frequency timer and a guard, this would result in a quite inefficient implementation with slightly different semantics.

Both EDS and SDL use abstractions decoupled from the hardware, and so any interactions with it have to be performed via a third-party component written in a native language. Memory, and so register access is done via function calls, through “glue-code”, which may be optimized to various extents, depending on the compiler and its settings. As a result, GPIO toggle timings were measured to be in the range between 7.5 and 75 microseconds, depending on the cache settings. While not especially large, they made it impossible to emulate custom protocols via “bit-banging” (in the original instrument design, instead of regular LEDs, digitally controlled NeoPixels were to be used). Interrupts are reported via sporadic interfaces, and so the reporting is done via first inserting the interrupt number into a queue, and then recovering it. While ~millisecond timings supporting UART communication were achieved, a direct implementation of interrupt handlers would be much more efficient. Additionally, memory consumption, requiring a few kilobytes of stack per each thread, created for each provided sporadic interface, while not significant for the use case and SAMV71 microcontroller, is considered higher than one that could be achieved via manual coding. This issue was painfully apparent in a similar project[3] when deploying a PolyORB based TASTE system onto STM32 microcontroller with 128 kB of usable RAM. While the currently designed system is larger, the used FreeRTOS based runtime is more streamlined (being derived from an optimized runtime targeting a 16-bit MSP430 [4]) and provides access to the 2 MB of external SDRAM, and so no issue needed to be resolved. However, this still needs to be considered for large and complex systems.

Conclusions and Future Work

It was demonstrated that it is possible to combine EDS models with a HW abstraction layer inside the TASTE ecosystem (including SDL), enhancing their usability. The presented approach could boost adoption of EDS in MBSE activities and embedded systems. However, while data and interface descriptions have proven to be easy and efficient to use, the definition of component implementations in the EDS format suffers from the lack of publicly available authoring and debugging tools, as well as the limitations of the standard that we have described in the chapter above. The used abstractions introduce overheads, however, it should be possible to resolve these issues by optimizing the runtimes and code generation tools, or by connecting with lower-level standards, especially if additional metadata or interfaces were available for the models.

References

1) TASTE portal. https://taste.tools/

2) Spacecraft Onboard Interface Services – XML Specification for Electronic Data Sheets. 876.0-B-1

3) Capella to TASTE MBSE bridge, M. Kurowski, A. Wójcik, H.P.de Koning, M. Perrotin et al., MBSE2020

4) Tiny Runtime to Run Model-Based Software on CubeSats, R. Babski, M. Kurowski, K. Grochowski, M. Perrotin, MBSE2020

5) FreeRTOS. https://www.freertos.org/