

EPiC Series in Computing

Volume 114, 2026, Pages 241–256

Proceedings of the Workshop on Mathematical Modeling and Scientific Computing



A Protocol-Agnostic, Human- and Machine-Readable ICD Format for Automated Software Generation, Validation, and Verification

Viktor Sinitsyn, Nils Schlautmann, and Florian Holzapfel

Institute of Flight System Dynamics, Technical University of Munich, Munich, Germany
viktor.sinitsyn@tum.de, nils.schlautmann@tum.de, florian.holzapfel@tum.de

Abstract

Interface Control Documents (ICDs) are commonly used to describe the exchange of data between distributed software and hardware components. However, in many engineering workflows they remain primarily human-readable documents, while code development, validation, verification, test configuration, and system-level simulation require additional manually maintained artifacts. This separation increases the risk of inconsistencies between functional software interfaces, transport-layer implementations, physical connections, and verification environments.

This work presents the concept of a protocol-agnostic, human- and machine-readable ICD format intended to serve as a controlled engineering artifact for automated software generation, validation, and verification. The proposed format builds on a previously developed unified message model for heterogeneous serial data exchange protocols and extends its practical use toward a machine-processable toolchain artifact.

The format is intended to link three complementary views of an interface: the logical view, describing application software inputs, outputs, and data structures; the transport view, describing message structures, encoding details, timing, and protocol-specific constraints; and the physical view, describing communication channels, physical links, and pinout assignments.

The paper also shows how such an ICD format can support the automation of several engineering tasks: generation of transport-layer software, generation of functional software interface data, generation of transport-layer configuration or code for HIL simulation environments, and construction of integrated simulated environment models composed of multiple device models and their interconnections.

The proposed approach is motivated by aerospace and embedded-system development, where heterogeneous communication protocols, distributed responsibilities, and strict verification requirements make consistency between documentation, implementation, and simulation artifacts a central engineering challenge. However, its applicability is not limited to aerospace systems. The same approach can be used in any domain where complex systems are composed of multiple interacting components that exchange data through numerous and heterogeneous communication protocols.

V. Turova, A. Kovtanyuk, A. Maslovskaya and J. Zimmer (eds.), MMSC 2026 (EPiC Series in Computing, vol. 114), pp. 241–256

1 Introduction and motivation

Interface Control Documents (ICDs) describe the interactions between system components. Although they are commonly used as inputs to subsequent software development, in many engineering workflows they remain primarily documents intended for manual interpretation. Information contained in an ICD is subsequently transferred into additional artifacts such as application software (hereafter referred to as "functional SW" or simply "functions") interface definitions, transport-layer software or configuration, test definitions, Hardware-in-the-Loop (HIL) configurations, and simulation models. When these artifacts are created or maintained independently, changes to an interface must be propagated across several representations, increasing both engineering effort and the risk of inconsistencies between documentation, implementation, and verification environments.

This work follows a different principle: the ICD is maintained as the single formal engineering artifact, while downstream interface-related artifacts are derived from it automatically whenever feasible. To support this role, the ICD must contain sufficient information not only to describe the data exchanged between components, but also to establish explicit relationships between application-level variables, their transport-layer representations, and the physical interfaces (also called "physical ports" in the paper) used for communication. At the same time, the representation remains textual and human-readable, allowing engineers to inspect relevant interface properties directly and to review modifications using conventional configuration-management tools such as Git.

The practical use of such an ICD within an automated airborne-software development workflow was demonstrated in our previous work, where an XML-based ICD was used to generate functional software interface definitions and transport-layer software [16]. That work focused on the overall development process and provided only a high-level description of the ICD format. In a separate study, we introduced the Unified Message Model (UMM), a protocol-agnostic model for describing the structure of messages exchanged over heterogeneous serial communication protocols [15]. The present work provides a detailed formalization of the complete ICD format in which the UMM forms the transport view and is explicitly linked to the logical and physical views.

A further motivation for emphasizing the scope and semantics of the stored information, rather than compatibility with a particular existing exchange format, is the decreasing practical cost of developing data translators and tool adapters. Modern software-development techniques, including AI-assisted implementation, make it increasingly feasible to connect well-defined formal representations to heterogeneous toolchains. Consequently, the key design question shifts from whether every target tool already supports a given format toward whether the format captures the information and relationships required to derive the necessary engineering artifacts reliably.

The main contributions of this paper are:

1. a formal ICD structure that separates interface information into distinct logical/application, transport, and physical views within a single engineering artifact;
2. incorporation of the previously proposed protocol-agnostic UMM as the transport-view model, enabling communication protocols with different message structures and encoding rules (hereafter referred to as heterogeneous communication protocols) to be represented within the same ICD architecture;
3. explicit binding mechanisms between the views, including mappings between logical parameters and transport containers and between transport contents and physical ports,

with support for differing logical and transport data types;

4. a machine-processable XML realization with an accompanying schema and representative examples, designed to serve as a single source of truth for automated generation, validation, verification, and simulation-related engineering artifacts.

2 Use cases and requirements for a formal ICD

The proposed ICD format is intended for workflows in which functional software, communication software, and physical integration can be treated separately. For the applications considered in this work, interface information is therefore divided into three layers: logical/application, transport, and physical. This is not intended to reproduce the complete OSI communication stack. Instead, it reflects the information needed in the model-based aerospace and embedded-system development workflows addressed in this paper.

The application view describes the inputs and outputs used by functional software. The transport view describes how these values are encoded into transmitted messages. The physical view describes communication ports, connectors, contacts, and related physical interface information.

A key design principle is to keep layer-specific details separated for developers. For example, functional software developers should work with engineering values and data types suitable for their development environment, rather than with bit positions, scaling factors, byte order, or protocol-specific encoding. In our current workflow, the main target environments are Simulink for functional SW and C for transport-layer SW.

The required conversion between application-level values and their transmitted representations is handled by the transport-layer software. The same separation between functional and transport-layer software is described in our previously published development workflow [16]. The practical value of separating these views has also been observed by Schlautmann et al. [13], who show that changes confined to the transport or physical view may leave the logical software interface unchanged and vice versa, allowing the corresponding engineering domains to evolve at different rates without unnecessary updates to unaffected artifacts.

The ICD format is based on four main design principles.

1. A single source of truth for automation. The ICD is intended to be the single maintained source of interface information, while other interface-related artifacts are derived from it automatically whenever possible. These artifacts may include functional software interface definitions, transport-layer software or configuration, test and verification data, HIL configurations, and simulation models. Several such applications are already part of the workflow described in [16]. For this purpose, the ICD must contain all information needed by the corresponding tools.

2. Separation of interface layers: Application, transport, and physical information are described separately but linked within the same ICD. In particular, an application parameter and its transmitted representation are treated as different entities. They may use different data types, and the ICD explicitly defines the mapping between them. This makes it possible to keep scaling, offsets, enumeration mappings, bit-level encoding, and similar conversions inside the transport layer.

3. Protocol-independent extensibility. Adding a new communication protocol should not require redesigning the ICD structure. New protocols may introduce new message types,

data types, attributes, or validation rules, but these additions should be made by defining new elements or reusing and combining existing elements within the defined model, rather than by changing the structure of the ICD. This allows the same ICD architecture to support heterogeneous communication technologies within one project.

4. Machine-processable and human-readable representation. The ICD must be formally processable by software, but it should also remain understandable to an engineer when inspected directly. In this paper, human-readable does not mean user-friendly. It means that properties such as message identifiers, parameter types, encoding definitions, and references can be understood from the textual representation. The same property also makes changes easy to review using standard version-control tools such as Git.

3 Related work and state of the art

Several established machine-readable formats address parts of the interface-description problem considered in this work. They differ, however, in scope, supported communication technologies, and in the way application-level data, transmitted representations, and physical interfaces are related.

DBC is widely used for CAN network descriptions [17]. It defines messages, signals, identifiers, bit positions, byte order, scaling, and related transport properties. Its scope is therefore both protocol-specific and comparatively narrow: it describes essentially the transport view of CAN communication and does not provide corresponding logical software-interface or physical-interface models.

AUTOSAR provides a substantially broader and mature system model with explicit separation and mapping of software, communication, and physical network information [2]. It also provides rich data-type and conversion models. Its communication descriptions, however, are based on explicitly supported automotive technologies and protocol-specific elements. Consequently, unrelated avionics or project-specific protocols cannot generally be introduced merely by defining new instances of a common protocol-independent message model. FIBEX has similar limitations relevant to the present work [1]: it provides a powerful machine-readable network description, but is automotive-oriented and extends its common model through technology-specific descriptions for individual communication protocols.

EUROCAE ED-247B addresses virtual and hybrid avionics integration and supports the transport of data associated with several avionics communication technologies [7]. However, ED-247 defines the organization of the virtualized exchange rather than the internal structure of the encapsulated avionics messages. The payload of the underlying protocol is transferred through ED-247 streams, while its signal-level composition remains external to ED-247 [4]. ED-247B can therefore complement an ICD-based toolchain, but by itself does not contain the detailed transport-message information required to generate protocol-specific serialization and deserialization software.

The Universal Configuration Format (UCoF) follows a concept closely related to the approach presented here. UCoF uses a model-based Single Source of Truth (SSoT) from which target-specific configuration artifacts are derived automatically [3, 5]. Recent work demonstrates its application to virtual and hybrid avionics test environments, including automatic generation of ED-247 gateway configurations [4]. These results independently support two central assumptions of the present work: interface-related information should be maintained in a common SSoT, and derived engineering artifacts should be generated from it rather than maintained independently. The published UCoF applications focus primarily on Integrated

Modular Avionics (IMA) / Distributed Integrated Modular Avionics (DIMA) platform and test-environment configuration, whereas the workflow considered here uses the same ICD information for both onboard-software and test-related artifacts. Furthermore, the current UCoF model represents individual communication technologies through protocol-specific model classes rather than through a common protocol-agnostic description of message structure [9]. The two approaches are therefore complementary: UCoF addresses a broader platform-configuration problem, while the UMM used in this work provides a detailed generic representation of heterogeneous and project-specific transport messages.

CCSDS 876.0-B-1 defines Spacecraft Onboard Interface Services Electronic Data Sheets (SEDS), a machine-readable format intended to describe onboard-device interfaces and to support, among other applications, automatic generation of onboard and simulation software [6]. SEDS provides rich data types, structured containers, explicit encodings, and mappings between interface levels, making it one of the approaches closest to the present work. Two differences are relevant here. First, parts of a protocol data unit that are defined by the underlying subnetwork protocol may remain outside the datasheet and are expected to be known by the corresponding toolchain. In contrast, the proposed ICD aims to represent the complete transport-message structure explicitly through the UMM. Second, transformations between SEDS interface levels may require component mappings, activities, or behavioural descriptions, whereas the proposed approach uses direct declarative links between logical parameters and transport containers, with representation conversion performed by generated transport-layer software.

General-purpose technologies address narrower parts of the problem. Protocol Buffers define structured data together with their own serialization mechanism [8], while Kaitai Struct can describe arbitrary binary layouts and low-level representation details [11]. They provide useful mechanisms for serialization or binary-format description, but do not provide the combined logical, transport, and physical engineering model required for the ICD considered here.

The approaches reviewed above demonstrate that many individual capabilities required for a formal ICD are already well established. The contribution of the present work is therefore not a claim that the proposed format is unique or universally applicable. Rather, it combines these capabilities with a protocol-agnostic Unified Message Model (UMM) [15], allowing standardized and project-specific message formats to be represented without protocol-specific extensions of the underlying ICD structure. The resulting ICD has also been integrated into a practical engineering toolchain and used as a common source for generation of both onboard-software and test-related artifacts, as discussed in Section 5.

4 Proposed protocol-agnostic ICD model

4.1 Overall structure and information model

The proposed ICD separates interface information into physical, logical, and transport views while representing them within a single formal artifact. The physical view describes communication ports and their electrical properties such as connections or baud rates, the logical view describes software functions and exchanged parameters, and the transport view describes the messages used to transfer these data between devices. The relationships between these views are represented explicitly and are discussed in the following subsections.

In the XML representation, device-specific information is organized under the *Device* element. Logical interface information is represented by *Functions* (application SW) and their *Parameters* (I/O structures), physical interfaces by *Ports*, and transport-level communication by *Port_Contents* containing individual *Frames*. Reusable definitions, including *DataTypes*

and *FrameTypes*, are stored separately and referenced by the corresponding model elements.

Figure 1 provides an overview of the principal entities and relationships of the proposed ICD model. The subsequent subsections describe the physical, logical, and transport views individually, followed by the mechanisms used to bind transport information to physical interfaces and logical parameters.

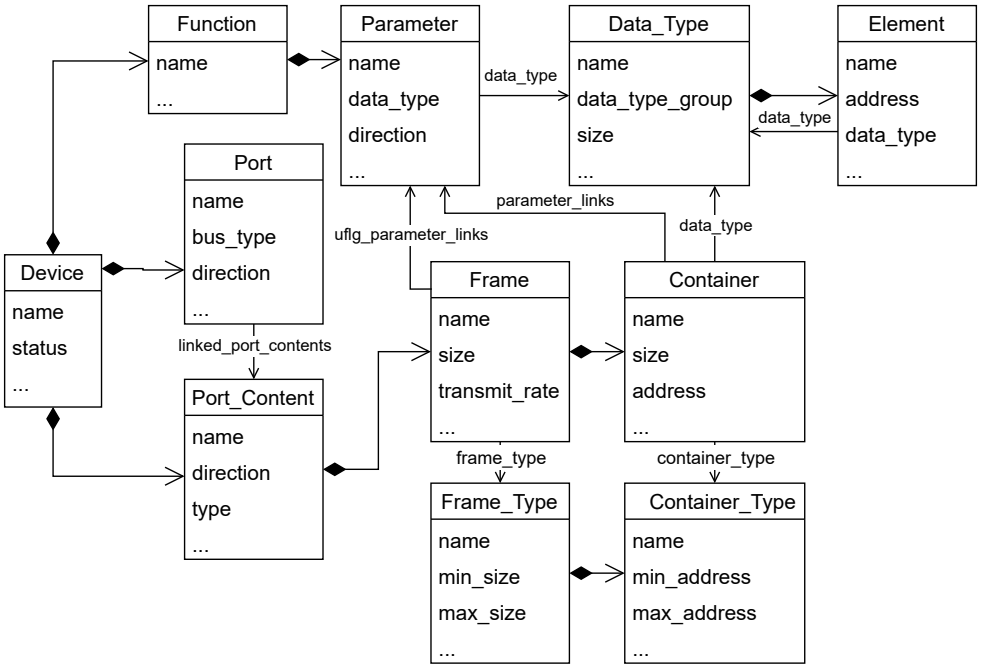


Figure 1: Simplified UML-style representation of the principal entities and relationships of the ICD model.

```
<root>
  <Devices>
    <Device ...>
      <Functions>
        <!-- ... -->
      </Functions>
      <Ports>
        <!-- ... -->
      </Ports>
      <Port_Contents>
        <!-- ... -->
      </Port_Contents>
    </Device>
  </Devices>
  <DataTypes>
    <!-- ... -->
  </DataTypes>
  <FrameTypes>
    <!-- ... -->
  </FrameTypes>
</root>
```

Listing 1: Simplified top-level structure of the XML representation.

Listing 1 shows a simplified top-level view of the XML serialization. Device-specific logical, physical, and transport information is grouped under the *Device* element, while reusable *DataTypes* and *FrameTypes* are maintained separately and referenced by the corresponding model elements.

Except for the simplified structural overview in Listing 1, all XML listings in this section are literal excerpts from the same reference ICD. The complete reference ICD and the corresponding XML Schema Definition (XSD) are provided in the accompanying ICDForge repository [10].

4.2 Physical interface model

The physical view of the ICD describes the physical interfaces of the device and their physical connection to the surrounding system. A physical interface is represented by a *Port* object. Each port defines its interface type and direction and may contain protocol- or implementation-specific attributes required to configure the corresponding physical resource. Physical connectivity is represented by the associated connector contacts.

The set of port attributes depends on the communication technology. For example, the CAN interface in the reference ICD specifies a baud rate of 250 kbit/s, whereas the RS-485 interface additionally specifies the number of data bits, parity, and stop bits. The same representation is therefore applicable to communication technologies requiring different sets of configuration parameters without changing the overall ICD structure.

Listing 2 shows the representation of an RS-485 communication port. The *bus_type* identifies the communication technology, while the *Attributes* collection contains its configuration parameters. The *Contacts* collection establishes the physical connection by identifying the *connector*, *contact*, and corresponding *wire* of each electrical signal.

```
<Port id="5092" name="DSRS485_A1" bus_type="SRS485_2WS" bus_type_id="151" direction="duplex">
    <Attributes>
        <Attribute name="Baud rate" value="115200"/>
        <Attribute name="Data bits" value="8"/>
        <Attribute name="Parity" value="N"/>
        <Attribute name="Stop bits" value="1"/>
    </Attributes>
    <Contacts>
        <Contact id="8015" wire="Hi" connector="J1" contact="D8"/>
        <Contact id="8016" wire="Lo" connector="J1" contact="D3"/>
        <Contact id="8017" wire="Sh" connector="J1" contact="BShl"/>
    </Contacts>
    <Linked_Port_Contents>
        <Port_Content id="vi8_5092"/>
        <Port_Content id="624"/>
    </Linked_Port_Contents>
</Port>
```

Listing 2: RS-485 port representation from the reference ICD.

4.3 Logical interface model

The logical view describes the application software-facing interface of the device independently of the communication mechanism used to exchange the corresponding data. Software interfaces are organized into *Functions*, each representing a separate software application and containing a set of input and output *Parameters*. A parameter specifies its direction and references a data type defining the structure of the exchanged value. Additional properties, such as the physical units (e.g. meters, radians, knots), may also be associated with the parameter.

Listing 3 shows the logical interface of the FlightControl function in the reference ICD. In this example, the function receives actuator-monitoring (IACTxx) and air-data structures (IADC)

and provides an actuator-command structure (OACTxxxCmds). The parameters themselves do not contain the complete structural definition of these data; instead, they reference reusable entries from the common data-type model.

```
<Function id="751" name="FlightControl" layer="Development">
    <Parameters>
        <Parameter id="23736" name="IACTxx" direction="in" data_type_id="1419" unit="1" layer="Development">
        </Parameter>
        <Parameter id="23737" name="IADC" direction="in" data_type_id="1269" unit="1" layer="Development">
        </Parameter>
        <Parameter id="23743" name="OACTxxxCmds" direction="out" data_type_id="1432" unit="1" layer="
     Development">
        </Parameter>
    </Parameters>
</Function>
```

Listing 3: Logical interface of the FlightControl function.

```
       <DataType id="1269" name="DADC_MADC_AD" data_type_group="Complex" size="288" sign="n/a" status="draft
     ">
            <Elements>
                <Element id="42649" name="Airspeed" address="0" data_type_id="1272"/>
                <Element id="42668" name="OAT_K" address="144" data_type_id="539"/>
                <Element id="42674" name="AOA" address="192" data_type_id="539"/>
                <Element id="42680" name="AOS" address="240" data_type_id="539"/>
            </Elements>
            <Mappings>
                <Mapping linked_environment="simulink" linked_data_type="DADC_MADC_AD_Bus"/>
            </Mappings>
       </DataType>

       <DataType id="1272" name="DADC_MADC_AD_Airspeed" data_type_group="Complex" size="144" sign="n/a"
     status="draft">
            <Elements>
                <Element id="42605" name="IAS_mDs" address="0" data_type_id="539"/>
                <Element id="42611" name="TAS_mDs" address="48" data_type_id="539"/>
                <Element id="42617" name="VS_mDs" address="96" data_type_id="539"/>
            </Elements>
            <Mappings>
                <Mapping linked_environment="simulink" linked_data_type="DADC_MADC_AD_Airspeed_Bus"/>
            </Mappings>
       </DataType>

       <DataType id="539" name="DMADCA825_Payload" data_type_group="Complex" size="48" sign="n/a" status="
     draft">
            <Elements>
                <Element id="7892" name="Parameter" address="0" data_type_id="14"/>
                <Element id="7893" name="Selftest_Passed" address="36" data_type_id="1"/>
                <Element id="7894" name="Initialization_Complete" address="37" data_type_id="1"/>
                <Element id="7895" name="Limited_Accuracy" address="38" data_type_id="1"/>
                <Element id="7896" name="Signal_Valid" address="39" data_type_id="1"/>
                <Element id="46024" name="Update_flag" address="40" data_type_id="8"/>
            </Elements>
            <Mappings>
                <Mapping linked_environment="simulink" linked_data_type="DMADCA825_Payload_Bus"/>
            </Mappings>
       </DataType>

         <DataType id="14" name="FLOAT" data_type_group="Floating-Point" size="32" sign="yes" offset="0"
     status="released">
            <Mappings>
                <Mapping linked_environment="simulink" linked_data_type="single"/>
            </Mappings>
       </DataType>
```

Listing 4: Selected data-type definitions illustrating hierarchical composition.

Data types may be primitive or complex. Complex types are composed recursively from typed elements, which allow hierarchical software interfaces to be represented without flattening them into individual scalar parameters. Each element identifies its relative address within the enclosing type and references another data type. The same data type definition can therefore be reused at different levels of the hierarchy and by multiple parameters. The data-type model and its use for protocol-agnostic message representation are described in detail in our previous work on the Unified Message Model (UMM) [15].

Listing 4 illustrates this hierarchical composition for the IADC parameter. Its type DADC_MADC_AD contains an Airspeed element whose type is itself complex. The latter groups several air-data quantities, each represented using the common type named "DMADCA825_Payload". The lowest-level elements ultimately reference primitive data types such as FLOAT. The *Mappings* element associates an ICD data type with the corresponding representation in a target software environment; in the reference implementation, complex types are mapped to Simulink bus definitions and primitive types to native Simulink data types.

4.4 Transport interface model

The transport view describes how data are represented and exchanged in communication messages. Transport information is organized into *Port_Contents*, each containing a set of *Frames* transmitted or received as part of the corresponding communication flow. A frame specifies properties such as its size, transmission rate, and frame type and contains a set of *Containers* representing individual regions of the transmitted message.

Containers define the position and representation of data within a frame. Depending on their role, they may represent identifiers, length fields, payload data, checksums, or other protocol-specific information. A container specifies its bit address, data type, byte and bit ordering, and, where required, additional properties defining the transport representation of the value. The underlying container and data type models are described in detail in our previous work on the Unified Message Model (UMM) [15].

```
<Frame id="39449" name="F221_09_Cmd_ACTCE" size="48" transmit_rate_ms="10" frame_type="DACT_VendX_PNY_T01A"
     frame_type_id="77">
    <Uflg_Parameter_Links>
        <Link linked_parameter_path="FlightControl.OACTxxxCmds.ACTCE_rad.Uflg"/>
    </Uflg_Parameter_Links>
    <IDs>
        <Container id="39452" name="Frame_ID" address="0" data_type_id="8" container_type_id="526" endianness
     ="Big-Endian" bit_order="Most Significant Bit First" value="221"/>
        <Container id="39450" name="Actuator_ID" address="8" data_type_id="8" container_type_id="523"
     endianness="Big-Endian" bit_order="Most Significant Bit First" value="9"/>
    </IDs>
    <Payload>
        <Container id="39453" name="A16_CmdACTCE" address="16" data_type_id="1575" size="16" offset="0"
     resolution = "0.0000625" container_type_id="527" endianness="Big-Endian" bit_order="Most Significant Bit
     First">
            <Parameter_Links>
                <Link linked_parameter_path="FlightControl.OACTxxxCmds.ACTCE_rad.Signal"/>
            </Parameter_Links>
        </Container>
    </Payload>
    <CRCs>
        <Container id="39451" name="CRC" address="32" data_type_id="11" container_type_id="524" endianness="
     Big-Endian" bit_order="Most Significant Bit First" extra_properties=""/>
    </CRCs>
</Frame>
```

Listing 5: Actuator command frame in the transport view.

The frame in Listing 5 consists of two identifier fields, one payload field, and a checksum. The *address* attribute specifies the position of each container within the frame, while *data_type_id* defines the representation of its value. For the payload container, the transport representation is a signed 16-bit fixed-point value with a *resolution* of 0.0000625. The *Parameter_Links* visible in the listing establish the relation to the logical view and are discussed separately in Section 4.6.

Containers are not restricted to primitive data types. A container may refer to a complex type composed of multiple nested elements. In that case, the container address defines the position of the complete structure within the frame, while the relative addresses defined by the referenced complex data type determine the positions of its atomic elements within the transmitted representation. The same recursive data-type model can therefore be used both for scalar transport fields and for structured payloads.

The structure of a particular frame is constrained by a reusable *FrameType*. A frame type defines the common structural rules that frames of a particular message format must follow. It specifies the allowed frame size and a set of *ContainerTypes*, which identify the semantic role of containers and constrain their permitted location within the frame.

Listing 6 shows the *FrameType* definition referenced by the actuator message above. The definition specifies the permissible positions of the frame identifier, actuator identifier, payload, optional additional identifier, and checksum. Such information can also be used during automated processing, for example to determine the valid address range of a payload container or to verify that a checksum is located in the prescribed portion of a frame.

```
<FrameType id="77" name="DACT_VendX_PNY_T01A" min_size_bits="48" max_size_bits="48">
    <ContainerTypes>
        <ContainerType id="526" name="Frame_ID" type="ID" byte_endianness="Big-Endian" bit_endianness="Most
     Significant Bit First" min_address_bit="0" max_address_bit="7" located_at_frames_end="no"
     implementation_comment=""/>
        <ContainerType id="523" name="Actuator_ID" type="ID" byte_endianness="Big-Endian" bit_endianness="Most
      Significant Bit First" min_address_bit="8" max_address_bit="15" located_at_frames_end="no"
     implementation_comment=""/>
        <ContainerType id="527" name="Payload" type="Payload" byte_endianness="Big-Endian" bit_endianness="
     Most Significant Bit First" min_address_bit="16" max_address_bit="31" located_at_frames_end="no"
     implementation_comment=""/>
        <ContainerType id="525" name="Extra_ID" type="ID" byte_endianness="Big-Endian" bit_endianness="Most
     Significant Bit First" min_address_bit="16" max_address_bit="31" located_at_frames_end="no"
     implementation_comment=""/>
        <ContainerType id="524" name="CRC" type="Checksum" byte_endianness="Big-Endian" bit_endianness="Most
     Significant Bit First" min_address_bit="32" max_address_bit="47" located_at_frames_end="no"
     implementation_comment=""/>
    </ContainerTypes>
</FrameType>
```

Listing 6: Frame-type definition referenced by the actuator command frame.

The same transport model is used for other communication technologies. For example, the reference ICD describes CAN extended frames using the same *Frame* and *Container* abstractions, while a different *FrameType* defines CAN-specific identifier, length, payload, and checksum regions (refer to frame type with *id*="42" in the example ICD). Protocol-specific differences are therefore represented through reusable frame, container, and data-type definitions rather than through changes to the overall ICD structure. Refer to [15] for detailed descriptions and examples.

4.5 Mapping transport content to physical interfaces

The physical and transport views are connected by explicitly assigning one or more *Port_Content* objects to a physical Port. The association is defined by the *Linked_Port_Contents* collection

of the port. This keeps the description of the physical port independent of the messages exchanged through it: a port defines the physical and communication properties of the interface, whereas a port content defines the corresponding set of transmitted or received frames.

The relationship is many-to-many. A single physical port may be associated with multiple port contents, for example to represent separate incoming and outgoing communication flows of a duplex interface. Conversely, the same port content may be assigned to multiple physical ports. This is particularly useful for equipment providing redundant communication outputs, where the same set of messages is transmitted through several physically independent interfaces without duplicating the transport-level definition.

In the reference ICD, the duplex RS-485 port DSRS485_A1 is linked to two port contents, vi8_5092 and 624, which represent input and output transport content, respectively.

```
<Linked_Port_Contents>
    <Port_Content id="vi8_5092"/>
    <Port_Content id="624"/>
</Linked_Port_Contents>
```

Listing 7: Binding of the RS-485 port to input and output transport content.

4.6 Mapping transport content to logical interfaces

The transport and logical views are connected by explicit parameter links. A *Parameter_Links* collection associates a transport container with a logical parameter or with an element of its hierarchical data structure. The referenced object is identified by a path starting from the corresponding function and parameter and, when required, continuing through nested data-type elements. Both the transport container and the referenced logical object may therefore represent either scalar or structured data.

The allowed link cardinalities follow a simple rule: one output may feed multiple inputs, whereas each input may have only one source in order to avoid conflicting assignments. Accordingly, an output parameter of a function may be linked to multiple outgoing containers, while each outgoing container may be linked to only one logical parameter. Conversely, an input parameter may receive data from only one incoming container, whereas one incoming container may provide the same data to multiple logical parameters. For complex input parameters, individual elements may be linked separately; in this case, the complex parameter as a whole must not be linked simultaneously.

This asymmetry allows the same produced value to be distributed to several consumers while ensuring that every receiving object has an unambiguous source. Listing 8 shows such a case for an actuator-position response: a single incoming payload container is linked to corresponding elements of the input parameters of the ”FlightControl” and ”RTAssurance” functional applications.

```
<Payload>
    <Container id="39519" name="A16_ActPosition_rad" address="16" data_type_id="1575" size="16" offset="0"
     resolution = "0.0000625" container_type_id="527" endianness="Big-Endian" bit_order="Most Significant Bit
     First">
        <Parameter_Links>
            <Link linked_parameter_path="FlightControl.IACTxx.ACTCx.ACTCE.ActPosition_rad"/>
            <Link linked_parameter_path="RTAssurance.IACTxx.ACTCx.ACTCE.ActPosition_rad"/>
        </Parameter_Links>
    </Container>
</Payload>
```

Listing 8: Incoming transport container linked to two logical consumers.

For outgoing data, the relation is constrained in the opposite direction. A logical output may be used by several outgoing transport containers, but every individual outgoing container has a single logical source. Listing 9 shows an actuator command container linked to the corresponding element of the *FlightControl* output parameter.

```
<Payload>
    <Container id="39453" name="A16_CmdACTCE" address="16" data_type_id="1575" size="16" offset="0" resolution
      = "0.0000625" container_type_id="527" endianness="Big-Endian" bit_order="Most Significant Bit First">
        <Parameter_Links>
            <Link linked_parameter_path="FlightControl.OACTxxxCmds.ACTCE_rad.Signal"/>
        </Parameter_Links>
    </Container>
</Payload>
```

Listing 9: Outgoing transport container linked to a logical source.

The logical and transport data types connected by a link can differ. In Listings 8 and 9, the transport container uses data type FPXX_STS (data_type_id=”1575”), which defines a signed fixed-point representation of a variable size. The container further specifies a *size* of 16 bits, an *offset* of zero, and a *resolution* of 0.0000625. The corresponding logical elements, ActPosition_rad and Signal, ultimately use the logical FLOAT data type.The linked data-type definitions and container properties provide the representation information required for conversion between the engineering value used by the software application and its representation in the communication message, while the link establishes the correspondence between the two. The separation of layers described above allows the software applications to operate on application-oriented data types and engineering values while serialization, scaling, and representation conversion are handled outside the functional application.

Besides the data represented by transmitted payload fields, the ICD also supports frame-level update flags *Uflg_Parameter_Links*. For an outgoing frame, the linked logical update flag controls transmission: if its value is zero, the frame is not transmitted; any non-zero value enables transmission. For an incoming frame, the transport layer sets the linked update flag to one when a frame satisfying the identification and validation conditions is received within a timeout interval derived from the frame refresh rate and resets it to zero otherwise. The same received-frame status may be distributed to several logical consumers.

```
<Uflg_Parameter_Links>
    <Link linked_parameter_path="FlightControl.OACTxxxCmds.ACTCE_rad.Uflg"/>
</Uflg_Parameter_Links>
```

Listing 10: Update flag controlling transmission of an outgoing frame.

```
<Uflg_Parameter_Links>
    <Link linked_parameter_path="FlightControl.IACTxx.ACTCx.ACTCE.SetPositionResponse_uflg"/>
    <Link linked_parameter_path="RTAssurance.IACTxx.ACTCx.ACTCE.SetPositionResponse_uflg"/>
</Uflg_Parameter_Links>
```

Listing 11: Update status distributed to logical parameters for an incoming frame.

4.7 XML representation and schema

The proposed ICD model is currently serialized in XML, which provides a structured, machine-processable representation while remaining sufficiently transparent for direct inspection by engineers and for review using conventional version-control tools such as Git. The XML structure reflects the principal entities and relationships of the information model described above, but the model itself is not tied to this serialization. Other serialization formats could therefore be used without changing the underlying logical, transport, and physical concepts.

The reference ICD used throughout this section is provided in the accompanying ICDForge repository [10]. It illustrates the combined representation of physical ports, logical software interfaces, transport contents, reusable data types, frame types, and the links between them.

An XML Schema Definition (XSD) is used to formally specify the allowed document structure, element hierarchy, attributes, value types, and selected referential constraints. However, as further discussed in Section 4.8, XSD validation complements, but does not replace, semantic validation of the engineering model. The complete XSD is provided together with the reference ICD in [10].

4.8 Model integrity: issues and control mechanisms

A formal ICD provides a common basis for automated generation and integration activities, but its usefulness depends on the integrity of the information it contains. Validation can generally be divided into two complementary groups: schema-level validation, which checks the correctness of the XML representation, and semantic validation, which checks the consistency of the engineering model represented by it.

XML Schema validation can enforce the document structure, mandatory elements and attributes, value types, enumerations, and selected referential constraints. Semantic validation addresses relationships and engineering constraints that cannot be expressed sufficiently in XSD. These include, among others, consistency of references and hierarchical parameter paths, compliance with parameter-link cardinality rules, compatibility of linked logical and transport data types, and consistency of container placement within frame boundaries.

Data-type conversion rules are an important class of project-level semantic constraints. The permitted conversions between logical and transport representations are defined during project planning and form the basis for the corresponding conversion functions implemented in the transport-layer software. Conversions outside this predefined set are not supported and must therefore be rejected during model validation.

Insufficient integrity control typically shifts errors to later stages of the toolchain. Depending on the inconsistency, code generation or software build may fail, while other problems may only become visible during software or system integration. The proposed format does not prescribe where semantic validation must be implemented; it may be performed during model authoring, ICD generation or import, or by an independent validation tool.

In the current implementation, as described in [16], dBricks [12] performs such integrity checks continuously while the engineering model is populated.

5 Automated engineering toolchain implementation example

The proposed ICD format has been implemented as an intermediate engineering artifact within the automated airborne-software development toolchain described in detail in our previous work [16]. The overall process is summarized in Fig. 2. Interface information originating from equipment documentation and system architecture is consolidated in a structured engineering model. From this model, device-specific ICD files are automatically generated, baselined and used as controlled inputs to subsequent development activities. A more recent repository-centered implementation by Schlautmann et al. [13] uses the same device-specific ICD XML as the controlled interface artifact exchanged between the system, functional, and embedded development domains and applies automated consistency checks to the derived artifacts.

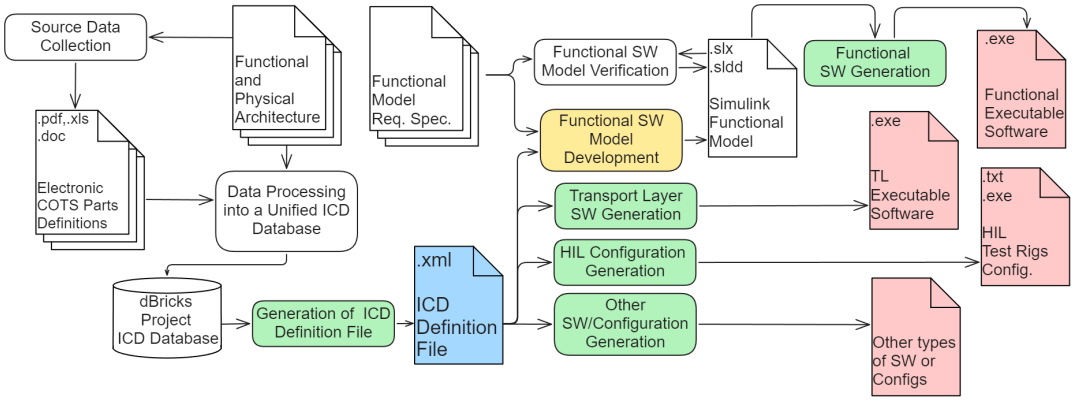


Figure 2: Role of the device-specific ICD within the automated engineering workflow and its use as a common input for software, HIL, and configuration generation.

For functional software development, the logical interface definitions contained in the ICD are used to automatically generate Simulink input/output structures and corresponding data definitions. Functional developers therefore work with application-oriented parameters and engineering values without manually reproducing interface definitions in the software models. The generated interface definitions can additionally be used as a reference for automated consistency checks against the developed functional models, as demonstrated by Schlautmann et al. [13].

In parallel, the same ICD provides the information required for transport-layer software generation. The transport-layer generator processes the physical and transport definitions together with the logical-to-transport mappings to generate software responsible for message reception and transmission, serialization and deserialization, and the required representation conversions. A concrete implementation of this approach is the tlcodegen toolbox described by Schwaiger [14], which generates transport-layer encode/decode implementations from ICDs based on the model described in this paper. Thus, the functional and transport-layer software are derived from a common interface definition rather than from independently maintained interface artifacts.

The same machine-readable ICD information can also be reused by other engineering activities. In the implemented toolchain, it is used for configuration of HIL interfaces and integrated simulation environments, and the same approach can be extended to further interface-dependent artifacts such as wiring-related engineering data.

The practical benefit of this approach is primarily the reduction of recurring manual syn-

chronization effort. Without automated derivation, a routine interface modification may require corresponding changes to functional-software interface definitions, transport-layer implementation, HIL or test-system configurations, and other interface-dependent artifacts. Maintaining these representations independently increases both the engineering effort associated with interface changes and the possibility of inconsistencies between them. In the implemented workflow, the interface modification is introduced into the common ICD and the affected downstream artifacts are regenerated or checked against this source.

In the proposed workflow, interface changes are intentionally managed at the system level rather than propagated back from downstream development tools. Interface information therefore flows from the system engineering model and the generated ICD to environments such as Simulink, transport-layer software development, and HIL or test-system configuration. If software developers require a change to an application interface, such as an additional variable or a modified data type, they request system engineers to introduce the change to the ICD. Once accepted, the affected interface definitions and other dependent artifacts can be regenerated automatically, so propagation of the change through the toolchain requires very little additional engineering effort.

6 Discussion and conclusion

This paper presented a formal ICD format that has been successfully used in several real-world engineering projects to support both human review and automated processing. We do not propose the presented format as a mandatory standard, but rather as a mature approach that may be considered when building automated engineering workflows. An important strength of the format is that it has evolved through practical use in several non-trivial engineering projects. Over this period, its principal concepts and overall structure have remained stable; extensions required by new use cases have generally been local additions to the model rather than replacements of its fundamental concepts.

The approach is also intended to be modular in its adoption. Users may adopt the complete format, reuse individual concepts such as the separation of logical, transport, and physical views, unified message model, or develop their own representations based on the same information model.

The presented ICD also provides a concrete implementation example of the unified message model introduced in our previous work [15], showing how that protocol-agnostic message model can be embedded into a broader ICD structure.

The essential contribution is not only a particular XML format, but also a definition of the data model that has proven useful for building automated interface-engineering toolchains. This distinction becomes increasingly relevant with the growing use of AI-assisted software development. The practical effort required to implement code generators, format converters, and tool adapters has decreased substantially, making integration with heterogeneous engineering tools less dependent on universal adoption of a single exchange format. Consequently, establishing a useful automation toolchain increasingly depends on understanding which information must be represented and how the engineering artifacts relate to each other, rather than on enforcing one rigid serialization or tool-specific format. The proposed ICD model provides one practically established answer to this information-modeling problem.

References

[1] ASAM e.V. ASAM MCD-2 NET (FIBEX): Data Model for ECU Network Systems. https://www.asam.net/standards/detail/mcd-2-net/. Version 4.1.2, released 2 June 2017; accessed: 2026-08-27.

[2] AUTOSAR. *AUTOSAR System Template*. AUTOSAR, classic platform release r23-11 edition, 2023.

[3] P. Chrysalidis, M. Halle, and F. Thielecke. Establishing a single source of truth for avionics platform verification through UCoF. In *2025 AIAA/IEEE 44th Digital Avionics Systems Conference (DASC)*, 2025.

[4] P. Chrysalidis, M. Halle, and F. Thielecke. A single-source-of-truth approach for automatic ED-247 gateway configuration in hybrid testing through UCoF. In *Software Engineering 2026 – Companion Proceedings*, pages 57–76. Gesellschaft für Informatik, 2026.

[5] P. Chrysalidis and F. Thielecke. A universal configuration format for avionics. In *Software Engineering 2024 Workshops*, pages 45–54, 2024.

[6] Consultative Committee for Space Data Systems. *Spacecraft Onboard Interface Services—XML Specification for Electronic Data Sheets*. CCSDS, issue 1 edition, April 2019.

[7] EUROCAE. *ED-247B: Technical Specification of Virtual Interoperable Simulation for Tests of Aircraft Systems in Virtual or Hybrid Bench*. EUROCAE, December 2023.

[8] Google. Protocol buffers: Overview. https://protobuf.dev/overview/. Accessed: 2026-08-27.

[9] Hamburg University of Technology, Institute of Aircraft Systems Engineering. UCoF: Universal configuration format. https://gitlab.com/ucof/ucof. Accessed: 2026-08-31.

[10] Institute of Flight System Dynamics, Technical University of Munich. Reference ICD and XML schema for "A protocol-agnostic, human- and machine-readable icd format for automated software generation, validation, and verification". https://gitlab.com/tum-fsd/icdforge, 2026. Release mmsc2026-paper-v1; accessed: 2026-10-26.

[11] Kaitai Project. Kaitai Struct User Guide. https://doc.kaitai.io/user_guide.html. Version 0.11; accessed: 2026-08-27.

[12] PEERSS GmbH. dBricks tool overview. https://peerss.net/dbricks/. Accessed: 2026-08-27.

[13] N. Schlautmann, V. Sinitsyn, B. Engelhard, and F. Holzapfel. From architecture to binary: Ensuring cross-domain consistency in model-based airborne software development, 2026. arXiv:2608.28156.

[14] F. Schwaiger. *Efficient Software Tools for Scalable Model-Based Validation and Verification*. PhD thesis, Technical University of Munich, Munich, Germany, 2026. 334 pages. Available via mediaTUM, record 1784761.

[15] V. Sinitsyn and F. Holzapfel. Unified message model for heterogeneous serial data exchange protocols, 2026. arXiv:2608.17642.

[16] V. Sinitsyn, N. Schlautmann, F. Schwaiger, and F. Holzapfel. Streamlined airborne software development for large UAVs: From unified data collection to automated code generation. In *DASC 2025 – Digital Avionics Systems Conference, Conference Proceedings*, pages 1–10. IEEE, 2025.

[17] Vector Informatik GmbH. DBC — database for CAN. https://elearning.vector.com/mod/glossary/showentry.php?eid=1734. Accessed: 2026-08-27.