INTERNATIONAL STANDARD

ISO/IEC 23001-4

Fourth edition
2017-08

Information technology — MPEG systems technologies —

Part 4: Codec configuration representation

Technologies de l'information — Technologies des systèmes MPEG — Partie 4: Représentation de configuration codec

Reference number
ISO/IEC 23001-4:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This fourth edition cancels and replaces the third edition (ISO/IEC 23001-4:2014), which constitutes a minor revision with the following changes:

— addition of citations to Annexes G, H and J in the Introduction;

— addition of a citation to Annex E in Clause 4;

— improvement of the usage description of `rvc:port attribute` and addition of a citation to Annex F in Clause 6;

— improvement of the specification of RVC-BSDL in Annex C;

— addition of informative description of a generic bitstream parser in Annex I.

A list of all parts in the ISO/IEC 23001 series can be found on the ISO website.

Introduction

This document defines the methods capable of describing codec configurations in the reconfigurable video coding (RVC) framework. The objective of RVC is to offer a framework that is capable of configuring and specifying video codecs as a collection of "higher level" modules by using video coding tools. The video coding tools are defined in the video tool library. ISO/IEC 23002–4 defines the MPEG video tool library. The RVC framework principle could also support non-MPEG tool libraries, provided that their developers have taken care to obey the appropriate rules of operation.

For the purpose of framework deployment, an appropriate description is needed to describe configurations of decoders composed of or instantiated from a subset of video tools from either one or more libraries. As illustrated in Figure 1, the configuration information consists of

— bitstream syntax description, and

— network of functional units (FUs) description (also referred to as the decoder configuration)

that together constitute the entire decoder description (DD).

Bitstreams of existing MPEG standards are specified by specific syntax structures and decoders are composed of various coding tools. Therefore, RVC includes support for bitstream syntax descriptions, as well as video coding tools. As depicted in Figure 1, a typical RVC decoder requires two types of information, namely the decoder description and the encoded media (e.g. video bitstreams) data.

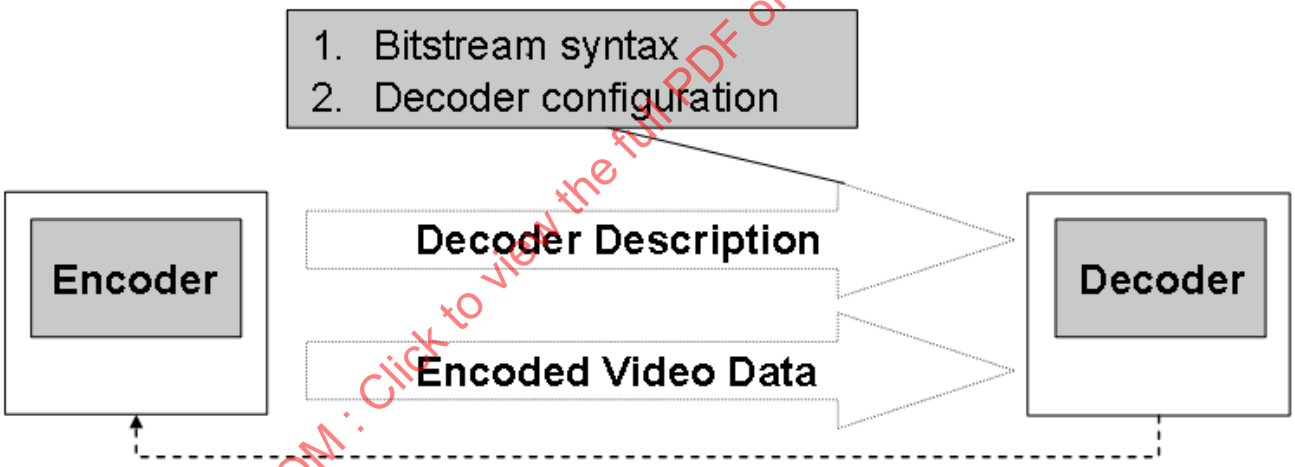

Figure 1 — Conceptual diagram of RVC

Figure 2 illustrates a more detailed description of the RVC decoder.

A more detailed description of the RVC decoder is shown in Figure 2, where the decoder description is required for the configuration of an RVC decoder. The Bitstream Syntax Description (BSD) and FU Network Description (FND) (which compose the Decoder Description) are used to configure or compose an abstract decoder model (ADM) which is instantiated through the selection of FUs from tool libraries optionally with proper parameter assignment. Such an ADM constitutes the behavioural reference model used in setting up a decoding solution under the RVC framework. The process of yielding a decoding solution may vary depending on the technologies used for the desired implementations. Examples of the instantiation of an abstract decoder model and generation of proprietary decoding solutions are given in Annex I.

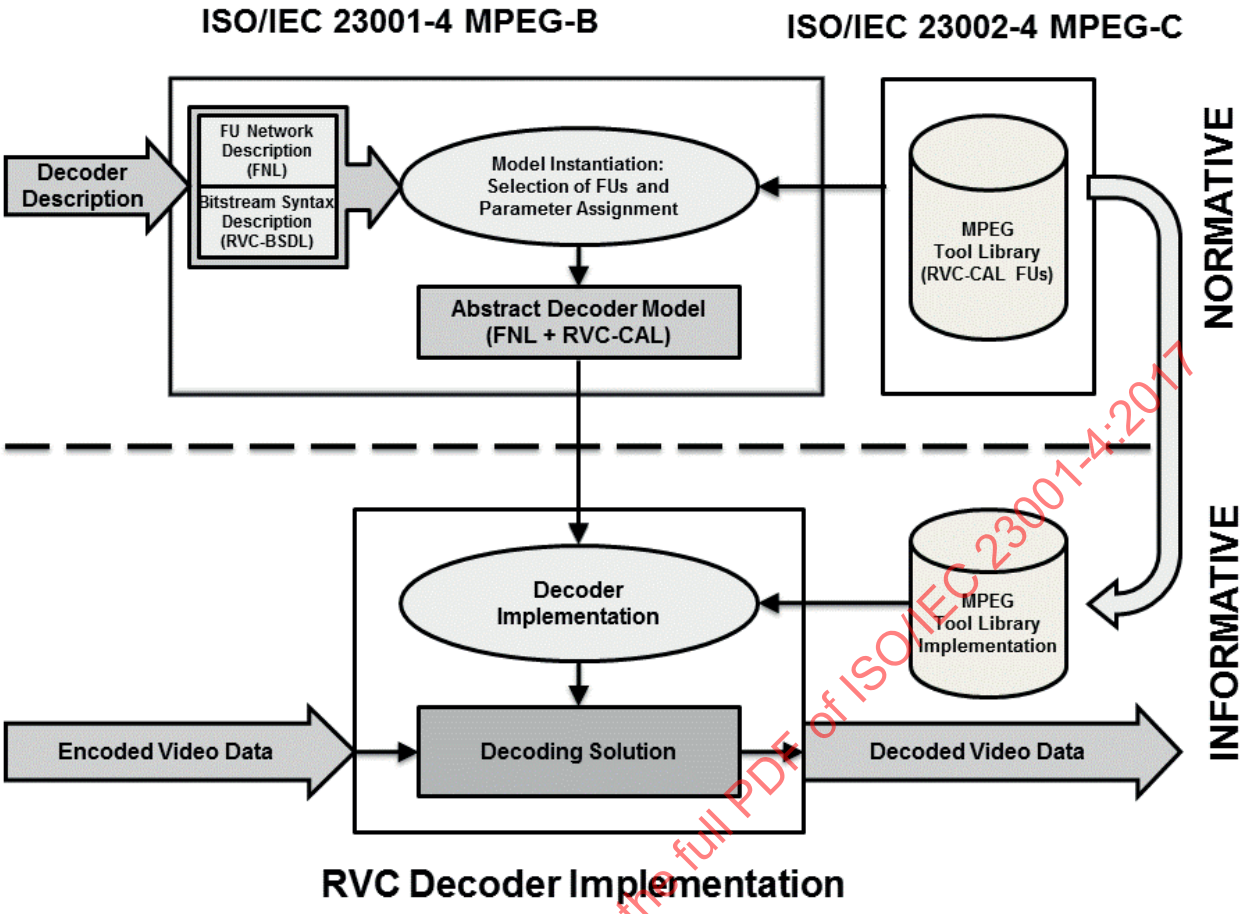

Figure 2 — Graphical representation of the instantiation process or decoder composition mechanism for the RVC normative ADM and for the non-normative proprietary compliant decoder implementation

Within the RVC framework, the decoder description describes a particular decoder configuration and consists of the FND and the BSD. The FND describes the connectivity of the network of FUs used to form a decoder whereas the parsing process for the bitstream syntax is implicitly described by the BSD. These two descriptions are specified using two standard XML-based languages or dialects:

— Functional Unit Network Language (FNL) is a language that describes the FND, known also as "network of FUs". The FNL specified normatively within the scope of the RVC framework is provided in this document;

— Bitstream Syntax Description Language (BSDL), standardized in ISO/IEC 23001-5 (MPEG-B Part 5), describes the bitstream syntax and the parsing rules. A pertinent subset of this BSDL named RVC-BSDL is defined within the scope of the current RVC framework. This RVC-BSDL also includes possibilities for further extensions, which are necessary to provide complete description of video bitstreams. RVC-BSDL specified normatively within the scope of the RVC framework is provided in this document.

The decoder configuration specified using FNL, together with the specification of the bitstream syntax using RVC-BSDL fully specifies the ADM and provides an "executable" model of the RVC decoder description.

The instantiated ADM includes the information about the selected FUs and how they should be connected. As already mentioned, the FND with the network connection information is expressed by using FNL. Furthermore, the RVC framework specifies and uses a dataflow-oriented language called

RVC-CAL for describing FUs' behaviour. The normative specification of RVC-CAL is provided in this document. The ADM is the behavioural model that should be referred to in order to implement any RVC conformant decoder. Any RVC compliant decoding solution/implementation can be achieved by using proprietary non-normative tools and mechanisms that yield decoders that behave equivalent to the RVC ADM.

The decoder description, the MPEG video tool library, and the associated instantiation of an ADM are normative. More precisely, the ADM is intended to be normative in terms of a behavioural model. In other words, what is normative is the input/output behaviour of the complete ADM, as well as the input/output behaviour of all the FUs that are included in the ADM.

This document also includes informative technical descriptions to facilitate implementation of the RVC framework. In Annex G, allocation of the decoder configuration data within MP4 file format is introduced. In Annex H, carriage of the decoder configuration over RTP is described. Finally, in Annex J, technical relation between the codec configuration representation and the MPEG multimedia middleware (M3W) is described.

Information technology — MPEG systems technologies —

Part 4: Codec configuration representation

1 Scope

This document defines the methods and general principles capable of describing codec configurations in the reconfigurable video coding (RVC) framework. It primarily addresses reconfigurable video aspects and will only focus on the description of representation for video codec configurations within the RVC framework.

Within the scope of the RVC framework, two languages, namely FNL and RVC-BSDL, are specified normatively. FNL is a language that describes the FND, also known as "network of FUs". RVC-BSDL is a pertinent subset of BSDL defined in ISO/IEC 23001-5. This RVC-BSDL also includes possibilities for further extensions, which are necessary to provide complete description of video bitstreams.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14496-2:2004, *Information technology — Coding of audio-visual objects — Part 2: Visual*

ISO/IEC 14496-12, *Information technology — Coding of audio-visual objects — Part 12: ISO base media file format*

ISO/IEC 23001-5, *Information technology — MPEG systems technologies — Part 5: Bitstream Syntax Description Language (BSDL)*

ISO/IEC 23002-4, *Information technology — MPEG video technologies — Part 4: Video tool library*

IETF RFC 1889, *RTP A Transport Protocol for Real-Time Applications, H. Schulzrinne, et. al., January 1996*

IETF RFC 2327, *SDP: Session Description Protocol, M. Handley, April 1998*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.1
abstract decoder model
ADM
conceptual model of the instantiation of *functional units* (3.8) from the *video tool library* (3.16) and their connection according to the *FU network description* (3.9)

3.2
bitstream syntax description
BSD
description containing the bitstream syntax, its implicit parsing rules and possibly tables [e.g. VLD tables if not already existing in the *reconfigurable video coding* (3.13) *video tool library* (3.16)] to define the parser *functional unit* (3.8)

Note 1 to entry: The BSD is expressed using reconfigurable video coding-bitstream syntax description language (3.14).

3.3
bitstream syntax description language
BSDL
description of the bitstream syntax and the parsing rules

Note 1 to entry: Bitstream syntax description language (BSDL) is standardized by ISO/IEC 23001-5.

3.4
connection
link from an output port to an input port of a *functional unit* (3.8) that enables token exchange between FUs

3.5
decoder configuration
conceptual configuration of a decoding solution

Note 1 to entry: Using the *MPEG video tool library* (3.12), decoder configuration can be designed as one of the following cases:

— a decoding solution of an existing MPEG standard at a specific profile and level;

— a new decoding solution built from tools of an existing MPEG standard;

— a new decoding solution built from tools of an existing MPEG standard and some new MPEG tools included in the MPEG video tool library;

— a new decoding solution that is composed of new MPEG tools included in the MPEG video tool library.

Note 2 to entry: In summary, an RVC decoder description essentially consists of a list of *functional units* (3.8) and of the specification of the FU connections [*FU network description* (3.9) expressed in *FU network language* (3.10)] plus the implicit specification of the parser in terms of *bitstream syntax description* (3.2) [BSD expressed in *reconfigurable video coding-bitstream syntax description language* (3.14)]. In order to be a complete behavioural model [i.e. *abstract decoder model* (3.1)] an RVC *decoder description* (3.6) needs to make reference to the behaviour of each FU that is provided in terms of I/O behaviour by the *MPEG video tool library* (3.12) specified in ISO/IEC 23002-4.

3.6
decoder description
DD
description of a particular decoder configuration, which consists of two parts: *FU network description* (3.9) and *bitstream syntax description* (3.2)

3.7
decoding solution
implementation of the *abstract decoder model* (3.1)

3.8
functional unit
FU
modular tool which consists of a processing unit characterized by the input/output behaviour

3.9
FU network description
FND
FU (3.8) connections used in forming a decoder which are modelled using *FU network language* (3.10)

3.10
FU network language
FNL
language that describes the *FU network description* (3.9), known also as a "network of FUs"

3.11
model instantiation
building of the *abstract decoder model* (3.1) from the *decoder description* (3.6) [consisting of the *bitstream syntax description* (3.2) and the *FU network description* (3.9)] and from *functional units* (3.8) from the *video tool library* (3.16)

Note 1 to entry: During the model instantiation, the parser FU is reconfigured according to the BSD or loaded from VTL.

3.12
MPEG video tool library
MPEG VTL
video tool library (3.16) that contains *functional units* (3.8) defined by MPEG, that is, drawn from existing MPEG International Standards

3.13
reconfigurable video coding
RVC
framework defined by MPEG to promote coding standards at tool-level while maintaining interoperability between solutions from different implementers

3.14
reconfigurable video coding-bitstream syntax description language
RVC-BSDL
pertinent subset of *bitstream syntax description language* (3.3), which is defined within the scope of the current *reconfigurable video coding* (3.13) framework

3.15
token
data entity exchanged between input and output among *functional units* (3.8)

3.16
video tool library
VTL
collection of *functional units* (3.8)

4 Functional unit network description

4.1 General

The FUs in MPEG RVC are specified by

— the textual description in ISO/IEC 23002-4, and

— the RVC-CAL reference software.

The RVC-CAL language is formally specified in Annex D, and the classification of FUs according to the dataflow computation model of RVC-CAL is informatively described in Annex E.

The Functional Unit Network Language (FNL) is specified in this subclause and is used to describe networks of FUs. FNL is derived from Extensible Markup Language (XML) which was in turn derived from SGML (ISO 8879). The ADM consists of a number of FUs with input and output ports, and the connections between those ports. In addition, the ADM may have input and output ports, which may be connected to the ports of FUs or to each other.

A decoder can be described as a network of a number of FUs or even only one FU (e.g. Figure 3).

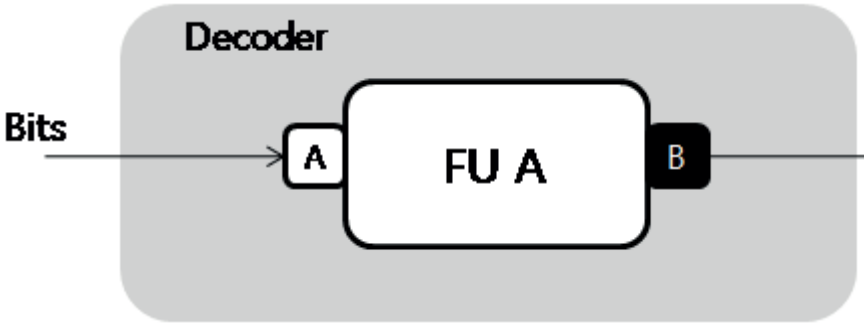

Figure 3 — FU network of one FU

A network of FUs is described in FND. An FND includes the list of selected FUs to form the decoder and the three types of connections: connections between FUs (type A), connections between decoder inputs and FU inputs (type B), and connections between FU outputs and decoder outputs (type C), which are illustrated in Figure 4.

The list of selected FUs (Figure 4) is described in FND according to Table 1. When selecting FUs from VTL, the IDs and names of FUs defined in ISO/IEC 23002-4 shall be used in the FND. The parameter assignments in the listed FUs are supported in the FND, but optional.

```
<Instance id="FU A">
         <Class name="Algo_Example1"/>
</Instance>
<Instance id="FU B">
         <Class name="Algo_Example2"/>
</Instance>
```

The connections (type A, type B, and type C shown in Figure 4) are described in FND as shown in Table 1.

Table 1 — Connection types

Type A	`<Connection src="FU A" src-port="B" dst="FU B" dst-port="D"/>` `<Connection src="FU A" src-port="C" dst="FU B" dst-port="E"/>`
Type B	`<input src="FU A" src-port="A"/>`
Type C	`<output src="FU B" src-port="F"/>`

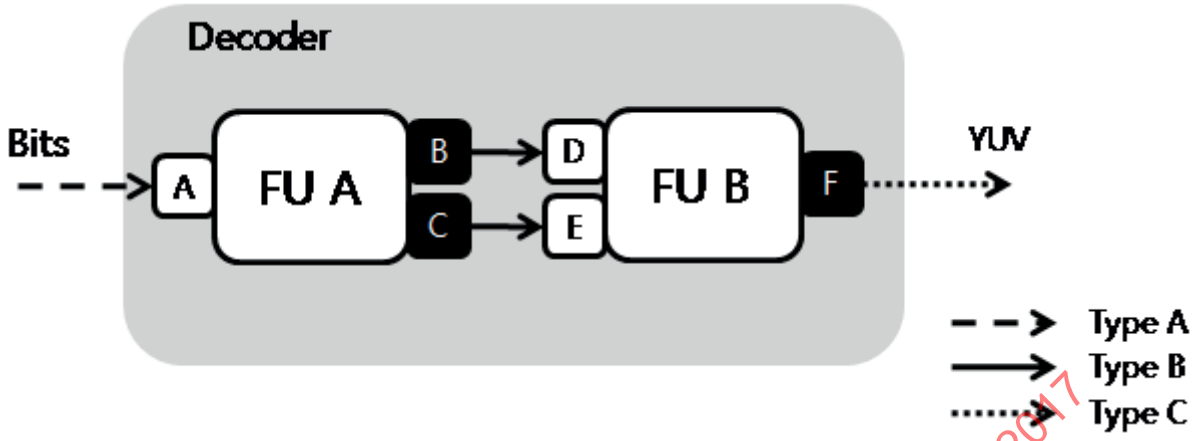

Figure 4 — Three types of connections in an FU network

Another example of FU networks with four FUs is illustrated in Figure 5. The textual description of Figure 5 in FND is described as follows.

```
<XDF name="Decoder">
<Instance id="Syntax parser">
   <Class name="syntax parser">
</Instance>
<Instance id="FU A">
   <Class name="Algo_ExamFU_A">
</Instance>
<Instance id="FU B">
   <Class name="Algo_ExamFU_B">
</Instance>
<Instance id="FU C">
   <Class name="Algo_ExamFU_C">
</Instance>
<Input src="Syntax Parser" src-port="A"/>
<Output src="FU C" src-port="R"/>
<Connection src="Syntax Parser" src-port="B" dst="FU A" dst-port="E"/>
<Connection src="Syntax Parser" src-port="C" dst="FU A" dst-port="F"/>
<Connection src="Syntax Parser" src-port="D" dst="FU B" dst-port="K"/>
<Connection src="FU A" src-port="H" dst="FU C" dst-port="O"/>
<Connection src="FU B" src-port="L" dst="FU C" dst-port="P"/>
<Connection src="FU B" src-port="M" dst="FU C" dst-port="Q"/>
</XDF>
```

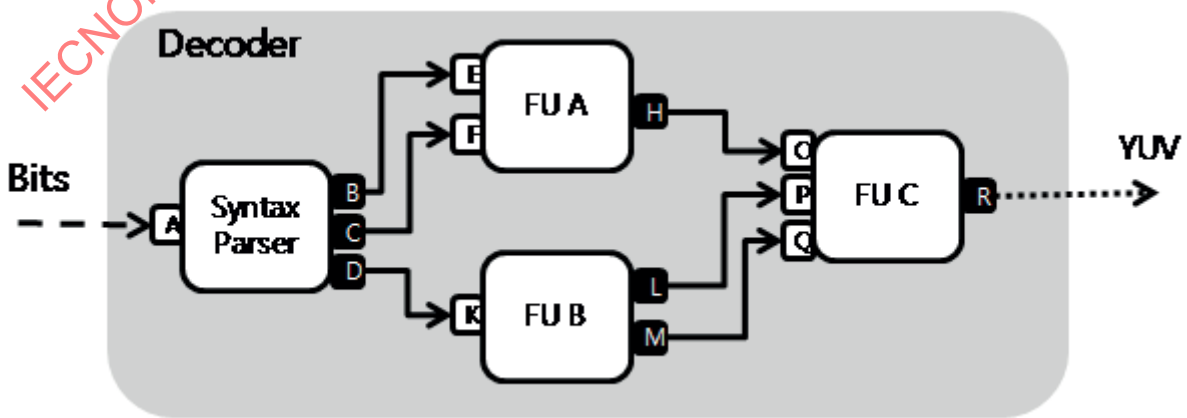

Figure 5 — Another example of FU networks

4.2 Specification of an FU network

The XML structures with names of elements, such as Decl, Network, Package, Expr, etc. are described in the specification of FNL in Annex A. In addition, attributes that direct an individual element's features are also introduced there. Attribute names will be prefixed with "@". For instance, common attribute names are @id, @name or @kind. In cases where an element name may be qualified by the value of an attribute, square brackets are used. For instance, in order to express the notion of an Expr element whose @kind attribute is the string "literal", Expr[@kind="literal"] is written.

By using the RVC-CAL model, FNL also allows FU networks and individual FUs to be parameterized. In particular, it is possible to pass bounded values for specific parameters into FU and FU networks. These values are represented by Expr and Decl syntax. Expr and Decl are the syntactical constructs describing a computation, which may, itself, be dependent upon the values of parameters which are either global or local variables.

5 Bitstream syntax description

The MPEG video tool library contains FUs that specify MPEG decoding tools. A new decoder configuration implies new bitstream syntax. The description of the bitstream syntax in RVC is provided using BSDL as specified in ISO/IEC 23001-5 and the BSDL schema. However, to facilitate the developments of synthesis tools that are able to generate parsers directly from a BSD (i.e. a BSDL schema), the RVC framework standardizes a version of BSDL called RVC-BSDL specified by including new RVC specific extensions and usage restrictions of standard BSDL in ISO/IEC 23001-5. Such extensions and restrictions versus the MPEG standard BSDL are defined in Annex C. RVC-BSDL contains all information necessary to parse any bitstream compliant with such syntax. The procedure to instantiate the parser capable of parsing and decoding any bitstream compliant with the syntax specified by the RVC-BSDL schema is not normative. Examples of such non-normative procedures are provided in Annex I.

6 Model instantiation

This clause describes the model instantiation process which consists of the selection of Functional Units (FUs) from the video tool library and instantiation of the FUs with the proper parameter assignments. The instantiation process requires the following information:

— video tool library;

— FU network description;

— bitstream syntax description.

The instantiation process consists of attaching the source code corresponding to the FUs identified in the FND in order to build a complete model that can be simulated. The video tool library is a library of source code of all FUs standardized in ISO/IEC 23002-4. The FND contains only the references (names of the FUs) to the pieces of code in the VTL. The process outputs the ADM. Figure 6 illustrates the model instantiation process.

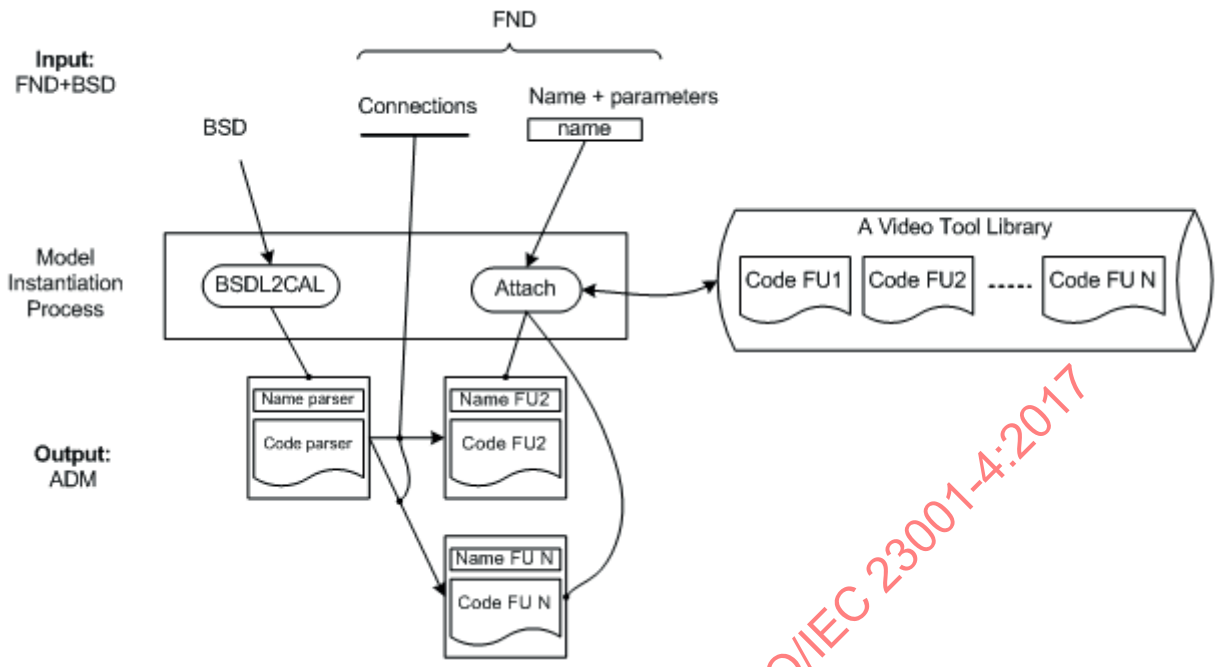

Figure 6 — Description of the model instantiation process

The FU Network Description (FND) provides the structure of the decoder by giving the names of the FUs composing the decoder and their respective connections among them. The name of the instance of the FU in the ADM is contained in the tag `<instance id="...">`. For instance, the attribute `rvc:port` (See C.4.4.5) indicates the name of the instance of the FU into the ADM to which this element of syntax is sent. The tag `<parameter>` provides the values of the parameters, which shall be used for the instantiation of the FU in the ADM.

The Bitstream Syntax Description (BSD) provides the structure of the bitstream. The parser is generated automatically from the BSD. Informative examples are provided in Annex I for building the parser. The syntax parser FU of the ADM might use other FUs to parse the bitstream. Thus, a clear link between identifiers inside the BSD and the FND shall be established. The tag `<rvc port="...">` indicates the name of the instance of the FU into the ADM to which this element of syntax is sent.

NOTE The FND could include instances of FUs that represent input and output to/from the network. Informative technical descriptions for the input/output FUs can be found in Annex F.

Annex A
(normative)

Functional unit network description

A.1 Elements of a functional unit network

A.1.1 XDF

An FU network is identified by the root element called XDF, which marks the beginning and end of the XML description of the network.

— Optional attribute: @name, the name of the network; @version, the version number of the current network (assumed to be "1.0" if absent).

— Optional children: Package, Decl, Port, Instance, Connection.

```
<XDF name="mpeg4SP">
     ...
</XDF>
```

A.1.2 Package

This element contains a structured representation of a qualified identifier (QID), i.e. a list of identifiers that are used to specify a locus in a hierarchical namespace. The QID provides the context for the @name attributed of the enclosing Network element: that name is intended to be valid within the specified namespace or package.

— Required child: QID, the qualified identifier.

```
<Package>
     <QID>
          <ID id="mpeg4"/>
     </QID>
</Package>
```

A.1.3 Decl[@kind="Param"]

This element represents the declaration of a parameter of the network.

— Required attribute: @name, the name of the parameter.

— Optional child: Type, the declared type of the parameter.

```
<Decl kind="Param" name="FOURMV"/>
```

A.1.4 Decl[@kind="Var"]

This element represents a variable declaration. Variables are used within expressions to compute parameter values for actors instantiated within the network and within expressions used to compute the values of other variables.

— Required attribute: @name, containing the name of the declared variable.

— Required child: Expr, representing the expression defining the value of this variable, possibly referring the values of other variables and parameters.

— Optional child: Type, the declared type of the variable.

```
<Decl kind="Variable" name="MOTION">
     <Expr kind="Literal" literal-kind="Integer" value="8"/>
</Decl>
```

A.1.5 Port

This element represents the declaration of a port of the network. Ports are directed, i.e. they serve as either input or output of tokens.

— Required attributes: @name, the name of the port. @kind, either "Input" or "Output".

— Optional children: Type, the declared type of the port.

```
<Port kind="Input" name="signed"/>
<Port kind="Output" name="out"/>
```

A.1.6 Instance

This element represents instantiations of FUs (i.e. actors). Essentially, an instantiation consists of two parts: (1) a specification of the class of the FU, and (2) a list of parameter values, specifying expressions for computing the actual parameter for each formal parameter of the FU class.

— Required attribute: @id, the unique identifier of this FU instance in the network. No two Instance elements may have the same value for this attribute.

— Required child: Class, identifying the FU class to be instantiated.

— Optional children: Parameter, each of these is assigning a formal parameter of the FU class to an expression defining the actual parameter value for this instance. Attribute, additional named attributes for this instance.

```
<Instance id="MPEG4_algo_PR">
     <Class name="MPEG4_algo_Add"/>
     <Parameter name="LAYOUT">
          <Expr kind="Literal" literal-kind="Integer" value="1"/>
     </Parameter>
</Instance>
```

```
<Instance id="Algo_IDCT2D_MPEGCPart1Compliant">
     <Class name="Algo_IDCT2D_MPEGCPart1Compliant"/>
</Instance>
```

A.1.7 Connection

This element represents a directed connection between two ports within the network. The source of the connection can be either an input port of the network or an output port of an FU instance. Conversely, the destination of that connection is either an output port of the network or the input port of an FU instance.

— Required attributes: @src, the id of the source FU of this connection. If it is not defined or set as a blank string (" "), the connection originates at a network input port. @src-port, the name of the source port. @dst, the id of the destination FU of this connection. If it is not defined or set as a blank

string (" "), the connection ends at a network output port. @dst-port, the destination port of the connection.

— Optional children: Attribute, additionally named attributes of this connection.

```
<Connection dst="MPEG4_algo_Add_V" dst-port="TEX"
src="Algo_IDCT2D_MPEGCPart1Compliant_V" src-port="out"/>
```

A.2 Expressions

A.2.1 General

All Expr elements represent expressions. Expressions are used to compute values that are in turn passed as parameters when instantiating FUs. Expressions can refer to variables by name. Those variables may be declared local variables of a network, declared network parameters, or global variables. There are a number of different kinds of expressions, all represented as Expr elements. They are distinguished by the @kind attribute.

A.2.2 Expr[@kind="Literal"]

These expressions represent literals, which are essentially atomic expressions that denote constants, and which do not refer to any variables. There are a number of different kinds of literals, distinguished by the @literal-kind attribute.

A.2.3 Expr[@kind="Literal"][@literal-kind="Boolean"]

These literals are Boolean values.

— Required attribute: @value, either "1" for true or "0" for false.

```
<Expr kind="Literal" literal-kind="Boolean" value="1"/>
```

A.2.4 Expr[@kind="Literal"][@literal-kind="Integer"]

These literals represent arbitrary-sized integral numbers.

— required attribute: @value, the decimal representation of the number.

```
<Expr kind="Literal" literal-kind="Integer" value="64"/>
```

A.2.5 Expr[@kind="Literal"][@literal-kind="Real"]

These are numbers with fractional parts.

— required attribute: value, the decimal representation of the number, optionally in scientific notation with an exponent separated from the mantissa by the character ‘E’ or ‘e’.

```
<Expr kind="Literal" literal-kind="Real" value="32e-2"/>
```

A.2.6 Expr[@kind="Literal"][@literal-kind="String"]

This expression represents are string literals.

— required attribute: @value, the string value.

```
<Expr kind="Literal" literal-kind="String" value="ForemanQCIF"/>
```


A.2.7 Expr[@kind="Literal"][@literal-kind="Character"]

This expression represents character literals.

— Required attribute: @value, the character value.

```
<Expr kind="Literal" literal-kind="Character" value="s"/>
```

A.2.8 Expr[@kind="List"]

This expression is a list.

```
<Expr kind="List"/>
```

A.2.9 Expr[@kind="Var"]

This expression is a variable reference.

— Required attributes: @name, the name of the variable referred to.

```
<Expr kind="Var" name="INTER"/>
```

A.2.10 Expr[@kind="Application"]

This expression represents the application of a function to a number of arguments.

— Required children: Expr, the expression representing the function. Args, an element containing the arguments.

```
<Expr kind="Application">
     <Expr kind="Var" name="log"/>
     <Args>
          <Expr kind="Literal" literal-kind="Integer" value="2"/>
     </Args>
</Expr>
```

A.2.11 Expr[@kind="UnaryOp"]

This expression represents the application of a unary operator to a single operand.

— Required children: Op, the operator. Expr, an expression representing the operand.

```
<Expr kind="UnaryOp">
     <Op name="!"/>
     <Expr kind="Literal" literal-kind="Boolean" value="1"/>
</Expr>
```

A.2.12 Expr[@kind="BinOpSeq"]

These expressions represent the use of binary operators on a number of operands. The associativity remains unspecified, and will have to be decided based on the operators involved. The children are operands and operators. There has to be at least one operator, and exactly one more operands than operators. The operators are placed between the operands in document order, with the first operator between the first and second operand, the second operator between the second and third operand and so forth. The relative position of operators and operands is of no importance.

— Required children: Op, the operators. Expr, the operands.


```
<Expr kind="BinOpSeq">
     <Expr kind="Literal" literal-kind="Integer" value="3"/>
     <Op name="+"/>
     <Expr kind="Literal" literal-kind="Integer" value="2"/>
</Expr>
```

A.3 Auxiliary elements

A.3.1 Args

This element contains the arguments of a function application.

— Required children: Expr, the argument expressions.

A.3.2 Op

This element represents a unary or binary operator, depending on context.

— Required attribute: @name, the operator name.

```
<Expr kind="Application">
   <Expr kind="Var" name="myfunction"/>
   <Args>
        <Expr kind="BinOpSeq">
             <Expr kind="Literal" literal-kind="Integer" value="3"/>
             <Op name="+"/>
             <Expr kind="Literal" literal-kind="Integer" value="2"/>
        </Expr>
   </Args>
</Expr>
```

A.4 Types

A.4.1 General

Types, represented by Type elements, occur in the declarations of variables and ports. They are used to specify the data types of objects bound to those variables or communicated via those ports. They are identified by a name, and may also comprise parameters, which are bound to either other types, or expressions (which are resulting in values).

A.4.2 Type

Type is the description of a data type.

— Required attribute: @name, the name of the type.

— Optional children: Entry, entries binding a concrete object (either a value or another type) to a named parameter.

```
<Type name="mytype">
     ...
</Type>
```

A.4.3 Entry[@kind="Expr"]

A value parameter of a type.

— Required attribute: @name, the name of the parameter.

— Required child: Expr, the expression used to compute the attribute value.

```
<Type name="mytype">
     <Entry kind="Expr" name="size">
          <Expr kind="Literal" literal-kind="Integer" value="10"/>
     </Entry>
</Type>
```

A.4.4 Entry[@kind="Type"]

A type parameter of a type.

— Required attribute: @name, the name of the parameter.

— Required child: Type, the type bound to the parameter.

```
<Type name="list">
     <Entry kind="Type" name="type">
          <Type name="bool"/>
     </Entry>
     <Entry kind="Expr" name="size">
          <Expr kind="Literal" literal-kind="Integer" value="32"/>
     </Entry>
</Type>
```

A.5 Miscellaneous elements

A.5.1 Attribute

The instances and connections of a network can be tagged with attributes. An attribute is a named element that contains additional information about the instance or connection. We distinguish four kinds of attributes: flags, string attributes, value attributes, and custom attributes. A flag is an attribute that does not contain ANY information except its name. A string attribute is one that contains a string, a value attribute contains an expression (represented by an Expr element), and a custom attribute contains any kind of information.

— Optional attribute: @value, the string value of a string attribute.

— Optional children: Expr, the value expression of a value attribute. An Attribute may instead also contain any other element.

```
<Connection dst="sink" dst-port="bits" src="source" src-port="bits">
     <Attribute kind="Value" name="bufferSize">
          <Expr kind="Literal" literal-kind="Integer" value="1"/>
     </Attribute>
</Connection>
```

A.5.2 QID

An element representing a qualified identifier, which is a list of simple identifiers. That list may be of any length, including zero.

— Optional children: ID, a simple identifier.

```
<QID>
     <ID id="mpeg4"/>
     <ID id="SP"/>
     <ID id="myversion"/>
</QID
```

A.5.3 FUID

A simple identifier.

— Required attribute: @id, the identifier.

```
<FUID id="0001012"/>
```

A.5.4 Class

This element identifies an FU class by name. If the FU class name is to be interpreted within a specific namespace, that QID of that namespace may be contained within the Class element.

— Required attribute: @name, the name of the class.

— Optional child: QID, the QID identifying the package/namespace for the class name.

```
<Class name="MPEG4_algo_VLDTableB8"/>
```

A.5.5 Parameter

This element specifies a value expression for a given, named parameter.

— Required attribute: @name, the parameter name.

— Required child: Expr, the expression whose evaluation will yield the value for the specified parameter.

```
<Parameter name="ROW">
     <Expr kind="Literal" literal-kind="Integer" value="1"/>
</Parameter>
```

NOTE This element is special in two respects: (1) It may occur anywhere in the network description. (2) Its format is entirely unspecified. The Note element can be used to add annotations and additional information to any element in the Network specification. It is common practice to use the @kind attribute to identify the type of the note.

Examples of description of networks of FUs using the FNL specified above are given in Annex B.

Annex B
(informative)

Examples of FU network description

B.1 Overview

This annex provides some examples of how an RVC decoder configuration can be specified in terms of a network of RVC FUs, including a 1-D IDCT, 2-D IDCT (Figure B.1) and the flatten MPEG-4 SP decoder FUs. A flatten decoder configuration is described in terms of networks of FUs from the RMC toolbox ISO/IEC 23002-4 composed of MPEG-4 SP FUs.

B.2 Example of specification of a network of FUs implementing a 1D-IDCT algorithm

Figure B.1 illustrates the network composed by five FUs taken from the MPEG RMC toolbox, the connections between FU and between the network and the outside world.

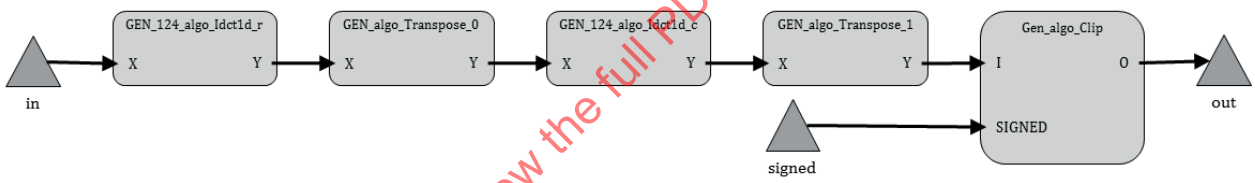

Figure B.1 — Example of networks of FU expressed using RVC FNL

The textual specification of the network in Figure B.1 is specified below. The network implements a 1-D IDCT.

```
<?xml version="1.0" encoding="UTF-8"?><XDF name="idct2d">

    <Package>
      <QID>
        <ID id="mpeg4"/>
      </QID>
    </Package>

   <Port kind="Input" name="in"/>
   <Port kind="Input" name="signed"/>
   <Port kind="Output" name="out"/>
   <Decl kind="Variable" name="INP_SZ">
      <Expr kind="Literal" literal-kind="Integer" value="12"/>
   </Decl>
   <Decl kind="Variable" name="PIX_SZ">
      <Expr kind="Literal" literal-kind="Integer" value="9"/>
   </Decl>
   <Decl kind="Variable" name="OUT_SZ">
      <Expr kind="Literal" literal-kind="Integer" value="10"/>
   </Decl>
   <Decl kind="Variable" name="MEM_SZ">
      <Expr kind="Literal" literal-kind="Integer" value="16"/>
   </Decl>
   <Instance id="GEN_124_algo_Idct1d_r">
      <Class name="GEN_124_algo_Idct1d"/>
      <Parameter name="ROW">
         <Expr kind="Literal" literal-kind="Integer" value="1"/>
   </Parameter>
   </Instance>
   <Instance id="GEN_algo_Transpose_0">
   <Class name="GEN_algo_Transpose"/>
   </Instance>
   <Instance id="GEN_124_algo_Idct1d_c">
   <Class name="GEN_124_algo_Idct1d"/>
   <Parameter name="ROW">
      <Expr kind="Literal" literal-kind="Integer" value="0"/>
      </Parameter>
   </Instance>
   <Instance id="GEN_algo_Transpose_1">
      <Class name="GEN_algo_Transpose"/>
   </Instance>
   <Instance id="GEN_algo_Clip">
      <Class name="GEN_algo_Clip"/>
      <Parameter name="isz">
         <Expr kind="Var" name="OUT_SZ"/>
      </Parameter>
      <Parameter name="osz">
         <Expr kind="Var" name="PIX_SZ"/>
      </Parameter>
   </Instance>
   <Connection dst="GEN_124_algo_Idct1d_r" dst-port="X" src="" src-port="in"/>
   <Connection dst="GEN_algo_Clip" dst-port="SIGNED" src="" src-port="signed"/>
   <Connection dst="" dst-port="out" src="GEN_algo_Clip" src-port="O"/>
   <Connection dst="GEN_algo_Transpose_0" dst-port="X" src="GEN_124_algo_Idct1d_r" src-port="Y"/>
   <Connection dst="GEN_124_algo_Idct1d_c" dst-port="X" src="GEN_algo_Transpose_0" src-port="Y"/>
   <Connection dst="GEN_algo_Transpose_1" dst-port="X" src="GEN_124_algo_Idct1d_c" src-port="Y"/>
   <Connection dst="GEN_algo_Clip" dst-port="I" src="GEN_algo_Transpose_1" src-port="Y"/>
</XDF>
```


B.3 FNL of the testbed

Figure B.2 — FNL of the testbed

```
<?xml version="1.0" encoding="UTF-8"?><XDF name="testbed">
  <Instance id="FUN_MPEG4SP_DECODER">
    <Class name="decoder"/>
  </Instance>
  <Instance id="fread">
    <Class name="fread"/>
    <Parameter name="fname">
      <Expr kind="Literal" literal-kind="String" value="data/foreman_qcif_30.bit"/>
    </Parameter>
  </Instance>
  <Instance id="DispYUV">
    <Class name="DispYUV"/>
    <Parameter name="title">
      <Expr kind="Literal" literal-kind="String" value="Foreman QCIF"/>
    </Parameter>
    <Parameter name="height">
      <Expr kind="Literal" literal-kind="Integer" value="144"/>
    </Parameter>
    <Parameter name="file">
      <Expr kind="Literal" literal-kind="String" value="data/foreman_qcif_30.yuv"/>
    </Parameter>
    <Parameter name="width">
      <Expr kind="Literal" literal-kind="Integer" value="176"/>
    </Parameter>
    <Parameter name="doCompare">
      <Expr kind="Literal" literal-kind="Integer" value="1"/>
    </Parameter>
  </Instance>
  <Connection dst="FUN_MPEG4SP_DECODER" dst-port="bits" src="fread" src-port="O"/>
  <Connection dst="DispYUV" dst-port="B" src="FUN_MPEG4SP_DECODER" src-port="VID"/>
</XDF>
```

Annex C
(normative)

Specification of RVC-BSDL

C.1 Overview

This annex describes the subset and the extensions of ISO/IEC 23001-5 BSDL that constitutes the specification of RVC-BSDL. The objective of specifying a new standard from BSDL (ISO/IEC 23001-5) into a smaller subset (RVC-BSDL), is to be able to support a simple and efficient methodology for describing video bitstreams syntaxes in the scope of RVC, as well as to facilitate the development of supporting tools (i.e. direct synthesis of parsers from RVC-BSDL descriptions).

The following subclauses describe the specificity of the subset and the extensions of BSDL standard as specified in ISO/IEC 23001-5, which are needed to obtain the RVC-BSDL used in this document (i.e. the RVC framework).

C.2 Use of prefixes in RVC-BSDL schema

Prefixes and the corresponding namespaces are specified in the RVC BSDL schema. Table C.1 shows the namespaces corresponding to each XML prefix. Because only a subset of XML or BSDL constructs is supported in RVC-BSDL, normative namespaces to define the construct subset are newly defined for RVC-BSDL. The namespaces of original standards are still compatible for RVC-BSDL description but are informative only for the backward compatibility.

Table C.1 — Mapping of prefixes to corresponding namespaces in RVC-BSDL schemas

Prefix	Corresponding Namespace (Normative)	Compatible Namespace (Informative)
xsd	urn:mpeg:mpegB:201x:RVC-BSDL-XSD-NS	http://www.w3.org/2001/XMLSchema
bs1	urn:mpeg:mpegB:201x:RVC-BSDL-BS1-NS	urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS
bs2	urn:mpeg:mpegB:201x:RVC-BSDL-BS2-NS	urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS
rvc	urn:mpeg:mpegB:201x:RVC-BSDL-RVC-NS	urn:mpeg:mpegB:2013:RVC-BSDL-RVC-NS

C.3 Constructs of RVC-BSDL

C.3.1 Overview

This subclause describes which XML or BSDL constructs are supported in RVC-BSDL in the RVC framework. It includes data types, attributes and elements. The aim of the subset definition is to provide a restricted way of representing well-defined bitstreams. Thus, the processes including the validations of the bitstreams and the generation of efficient implementations capable of parsing the bitstreams, described using RVC-BSDL, become simpler. The specification of the BSDL constructs listed below can be found in ISO/IEC 23001-5. The constructs that are not described in this subclause should not be considered to be supported in the RVC-BSDL syntax.

C.3.2 Supported data types

C.3.2.1 Built-in data types

This subclause describes the data types which are supported by RVC-BSDL. The supported data types already defined in common XML schema is shown in Table C.2. The BSDL built-in data types supported by RVC-BSDL are reported in Table C.3.

Table C.2 — List of XML Schema data types supported by RVC-BSDL

Data Type	Supported by RVC-BSDL
`xsd:hexBinary`	Yes
`xsd:long`	Yes
`xsd:int`	Yes
`xsd:short`	Yes
`xsd:byte`	Yes
`xsd:unsignedLong`	Yes
`xsd:unsignedInt`	Yes
`xsd:unsignedShort`	Yes
`xsd:unsignedByte`	Yes

Table C.3 — List of BSDL built-in data types supported by RVC-BSDL

Data Type	Supported by RVC-BSDL
`bs1:byteRange`	Yes
`bs1:align32`	Yes
`bs1:align16`	Yes
`bs1:align8`	Yes
`bs1:b1-bs1:b32`	Yes

C.3.2.2 Additional data types

Table C.4 shows the additional data type in RVC-BSDL.

Table C.4 — List of additional data type in RVC-BSDL

Data Type	Described in
`rvc:ext`	C.4.3.2.1

C.3.3 Supported elements

This subclause describes which BSDL facets are supported in RVC-BSDL within the RVC framework. No BSDL-1 elements are supported in RVC-BSDL. The allowed BSDL-2 elements are described in Table C.5. The allowed XML built-in elements are reported in Table C.6.

Table C.5 — BSDL-2 elements supported by RVC-BSDL

Element name	Supported by RVC-BSDL?
`bs2:bitLength`	Yes (see C.4.3.10)
`bs2:startCode`	Yes (see C.4.3.11)
`bs2:ifUnion`	Yes (see C.4.3.12)
`bs2:variable`	Yes (see C.4.3.6)

Table C.6 — XML standard elements supported by RVC-BSDL

Element name	Supported by RVC-BSDL?
`xsd:schema`	Yes (see C.4.3.1)
`xsd:sequence`	Yes (see C.4.3.4)
`xsd:choice`	Yes (see C.4.3.5)
`xsd:group`	Yes (see C.4.3.3)
`xsd:element`	Yes (see C.4.3.2)
`xsd:simpleType`	Yes (see C.4.3.7)
`xsd:annotation`	Yes (see C.4.3.8)
`xsd:appinfo`	Yes (see C.4.3.8)
`xsd:union`	Yes (see C.4.3.12)

C.3.3.1 Additional elements

Table C.7 shows the additional data type in RVC-BSDL.

Table C.7 — List of additional data type in RVC-BSDL

Element name	Described in
`rvc:allocation`	C.4.3.13

C.3.4 Supported attributes

C.3.4.1 Built-in attributes

This subclause describes which attributes are supported by RVC-BSDL within the RVC framework. No BSDL-1 attributes are supported in RVC-BSDL. The allowed BSDL-2 attributes are described in Table C.8. The allowed built-in XML attributes are described in Table C.9.

Table C.8 — BSDL-2 attributes supported by RVC-BSDL

Attribute name	Supported by RVC-BSDL?
`bs2:nOccurs`	Yes (See C.4.4.2)
`bs2:if`	Yes (See C.4.4.1)
`bs2:position`	Yes (See C.4.4.3)
`bs2:partContext`	Yes (See C.4.3.6)
`bs2:bsdlVersion`	Yes (See C.4.3.1)

Table C.9 — XML attributes supported by RVC-BSDL

Attribute name	Supported by the RVC-BSDL?
`fixed`	Yes
`name`	Yes
`value`	Yes

C.3.4.2 Additional attribute

Table C.10 shows the additional attributes in RVC-BSDL.

Table C.10 — List of additional attributes in RVC-BSDL

Attribute name	Described in
`rvc:iterator`	C.4.4.2
`rvc:iteratorInit`	C.4.4.2
`rvc:extName`	C.4.4.4
`rvc:extParams`	C.4.4.4
`rvc:port`	C.4.4.5
`rvc:rootGroup`	C.4.4.6
`rvc:isArray`	C.4.4.7
`rvc:dimension`	C.4.4.7

C.4 Syntax of RVC-BSDL

C.4.1 Overview

This subclause fully specifies the syntax of RVC-BSDL used in the context of the RVC framework. The allowed combinations of the elements, data types and attributes are reported in this subclause. It defines the subset RVC-BSDL.

C.4.2 Conventions

C.4.2.1 To define the syntax of the elements

— The attributes or children elements, which are shown in italic, are optional.

— The (a | b | c) construction means that one can choose only one element among a, b or c.

— The {a, b, c} construction means that one can build a list of several elements among a, b or c.

C.4.2.2 To define the syntax of the expressions

We use a form of BNF to describe the syntax rules. Literal elements are put in quotes (in the case of symbols and delimiters), or set in boldface (in the case of keywords). An optional occurrence of a sequence of symbols *A* is written as [*A*], while any numbers of consecutive occurrences (including none) are written as {*A*}. The alternative occurrence of either *A* or *B* is expressed as *A* | *B*.

We often use plural forms of non-terminal symbols without introducing them explicitly. These are supposed to stand for a comma-separated sequence of at least an instance of the non-terminal; e.g. if A is the non-terminal, we might use As in some production, and we implicitly assume the following definition: As → A { ‘,’ A }.

In the examples reported here, the usual interpretation of expression literals and mathematical operators is assumed, even though strictly speaking these are not part of the language and depend on the environment. A specific implementation of RVC-CAL may not have these operators, or interpret them on the other hand in a different manner.

C.4.3 Syntax of the elements

This subclause describes the syntax of the XML elements within the RVC-BSDL grammar.

C.4.3.1 xsd:schema **element**

This element is the top level element of RVC-BSDL schema description.

Syntax:

```
<xsd:schema
     bs2:bsdlVersion="ISO/IEC 23001-4:2014"
     rvc:rootGroup="string"
>
Children: {xsd:group, xsd:simpleType, xsd:annotation}
</xsd:schema>
```

Semantics:

The `bs2:bsdlVersion` attribute shall be annotated to specify the version of RVC-BSDL used for the current BSD. The value of the `bs2:bsdlVersion` attribute should be `"ISO/IEC 23001-4:2014"`.

The `rvc:rootGroup` attribute can be used to specify an `xsd:group` element that describes the top-level bitstream hierarchy group within the given BSD. Different from BSDL, RVC-BSDL does not allow `xsd:element` as a top-level construct of the bitstream syntax. Therefore, a `bs2:rootElement` attribute, which is a BSDL-2 attribute to designate the top-level bitstream hierarchy is not applicable in RVC-BSDL. As a replacement, RVC-BSDL defines `rvc:rootGroup` to designate the top level of a bitstream syntax. The bitstream syntax parser FU instantiated from the given BSD should start its bitstream parsing process from the top-level hierarchy group. When the `rvc:rootGroup` attribute is not defined, the `xsd:group` element that first appears on the RVC-BSDL schema should be considered as the top-level hierarchy.

For compatibility and authoring purposes, attributes specified in the XML schema recommendation (e.g. XML namespace designation) can be used within an `xsd:schema` element.

Example:

The example below shows a typical RVC-BSDL schema declaration using `xsd:schema`.

```
<xsd:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="urn:mpeg:mpegB:2014:RVC-BSDL-XSD-NS"
   xmlns:bs1="urn:mpeg:mpegB:2014:RVC-BSDL-BSDL1-NS"
   xmlns:bs2="urn:mpeg:mpegB:2014:RVC-BSDL-BSDL2-NS"
   xmlns:rvc="urn:mpeg:mpegB:2014:RVC-BSDL-NS"
   xmlns:m4v="urn:mpeg:mpeg4:profile:visual:simple"
   xsi:schemaLocation="
      urn:mpeg:mpegB:2013:RVC-BSDL-XSD-NS    ./n_xsd/MPEGB-RVC-BSDL-XSD.xsd
      urn:mpeg:mpegB:2013:RVC-BSDL-BSDL1-NS ./n_xsd/MPEGB-RVC-BSDL-BS1.xsd
      urn:mpeg:mpegB:2013:RVC-BSDL-BSDL2-NS ./n_xsd/MPEGB-RVC-BSDL-BS2.xsd
      urn:mpeg:mpegB:2013:RVC-BSDL-NS        ./n_xsd/MPEGB-RVC-BSDL-RVC.xsd
   bs2:bsdlVersion="ISO/IEC 23001-4:2014"
   rvc:rootGroup="bitstream">
```

C.4.3.2 xsd:element element

This element is used to define an element of syntax. A syntax element defined by this element can be one of the following cases:

— a fixed-length bitstream syntax element that is one of the fixed-length bit types derived from BSDL-1 (e.g. `bs1:b8`);

— a variable-length bitstream syntax element that is one of the user-defined types declared within the same BSD using the `xsd:simpletype` element (see C.4.3.7);

— a variable-length bitstream syntax element that is defined as `rvc:ext` and to be parsed by an external FU or a plug-in function in the parser (see C.4.3.2.1);

— a placeholder syntax element that is not actually derived from the bitstream. Placeholder syntax elements without type attribute can be used to contain `bs2:variable` elements to define necessary data calculation process or port output behaviours during the bitstream parsing process (see C.4.3.6).

Syntax:

```
<xsd:element
     name="string"
     type="(bs1:b1 - bs1:b32 | rvc:ext | bs1:align8 | bs1:align16 | bs1:align32
| user-defined type)"
     bs2:partContext="(true | false)"
     bs2:position="Expression{, Expression}"
     bs2:if="Expression"
     bs2:ifNext="NumericLiteral(, NumericLiteral)"
     bs2:nOccurs="(Expression | unbounded)"
     rvc:iterator="Expression"
     rvc:iteratorInit="Expression"
     value="Expression"
     fixed="Expression"
     rvc:port="string"
     rvc:extName="string"
     rvc:extParams="Expression">
Children: xsd:annotation
</xsd:element>
```

Semantics:

The `name` attribute defines the name of the syntax element. When the `bs2:partContext` attribute is true, the `name` attribute is also used as the name of an internal variable where the parsed data is stored.

The `type` attribute defines the type; in other words, the length of the syntax element. The bitstream parser FU should perform bitstream parsing according to the given type of the syntax element. Use of user-defined type defined by `xsd:simpleType` element is also allowed. If the `type` attribute is not defined, no data is read nor parsed from the bitstream.

When the `type` attribute is set to `rvc:ext`, the syntax element shall be parsed by an external FU or a plug-in function. See C.4.3.2.1 for the details.

The `value` attribute can be used to define post-processing of the value read from the bitstream. Within the expression, the `last()` function (see C.4.5.1) is used to represent the value read from the bitstream. This attribute can conveniently be used to describe syntax elements which value is not encoded as-is in the bitstream.

The `fixed` attribute forces a specific value to the given syntax element. When the value read from the bitstream is different from the value defined by the `fixed` attribute, the syntax parser FU may consider this as an error. This functionality can be used to define syntax element with predefined value (e.g. marker bit).

The `rvc:port` attribute defines the name of the output port where the data is parsed from the current syntax element to be exported.

If the `bs2:partContext` attribute is set to `"true"`, the parsed data should be stored in an internal variable by the syntax parser FU. Additionally, if the `bs2:position` attribute is set, the internal variable should be an array-typed variable, and the array index is given by the `bs2:position` attribute. See C.4.4.3 for the syntax of the `bs2:position` attribute.

About the attributes describing conditional statements, see C.4.4.1 (`bs2:if`) and C.4.4.2 (`bs2:nOccurs`, `rvc:iterator`, and `rvc:iteratorInit`) for their syntax and semantics.

Example:

The example below shows some typical `xsd:element` declarations.

```
<xsd:element name="sps_video_parameter_set_id" type="bs1:b4"/>
<xsd:element name="sps_max_sub_layers_minus1" type="bs1:b3"
  bs2:partContext="true"/>
<xsd:element name="sps_temporal_id_nesting_flag" type="bs1:b1"/>

 […]

<xsd:element name="sps_seq_parameter_set_id" type="rvc:ext"
  rvc:extName="EF_EXP_GOLOMB_U" bs2:partContext="true">
  <xsd:annotation><xsd:appinfo>
    <bs2:variable name="sps_id" value="sps_seq_parameter_set_id"/>
  </xsd:appinfo></xsd:annotation>
</xsd:element>

 […]

<xsd:element name="sps_chroma_format_idc" type="rvc:ext"
  rvc:extName="EF_EXP_GOLOMB_U" bs2:partContext="true" bs2:position="sps_id"/>
```

The example below shows a placeholder syntax element. This element is declared to describe variable control and port output behaviour without reading bitstream.

```
<xsd:element name="init_token">
 <xsd:annotation>
    <xsd:appinfo>
       <bs2:variable value="vop_quant" rvc:port="P_QP"/>
       <bs2:variable name="ac_coded" value="CBP[block_id] == 1"
         rvc:port="P_ACCODED"/>
       <bs2:variable value="ac_pred_flag" rvc:port="P_ACPRED"/>
       <bs2:variable value="fourmv" rvc:port="P_FOURMV"/>
       <bs2:variable value="0" rvc:port="P_MOTION" bs2:if="btype_intra"/>
       <bs2:variable value="2" rvc:port="P_BTYPE"  bs2:if="btype_intra"/>
       <bs2:variable value="1" rvc:port="P_MOTION" bs2:if="not btype_intra"/>
       <bs2:variable value="1" rvc:port="P_BTYPE"  bs2:if="not btype_intra"/>
    </xsd:appinfo>
 </xsd:annotation>
</xsd:element>
```

The example below shows a usage of value post-processing using the `value` attribute. The syntax element is encoded as `vps_max_sub_layers_minus1` in the bitstream. The inline post-processing expression described in the `value` attribute recovers the actual semantic value, `vps_max_sub_layers`, by simple addition. Such operation can be performed without declaring a `bs2:variable` element.

```
<xsd:element name="vps_max_sub_layers" type="bs1:b3" bs2:partContext="true"
     value="last()+1"/>
```

C.4.3.2.1 rvc:ext data type

It may happen that processing tasks associated to the parsing of a segment of the bitstream are not described in the RVC-BSDL schema. This is the case for bitstream segments for which VLD, CAVLD or CABAC decoding algorithms need to be applied. The data type `rvc:ext` indicates a portion of bitstream that needs to be decoded by an externally defined algorithm. The `rvc:ext` can be used to define the following cases of externally defined algorithm: (1) specific FUs available in the RMC toolbox, and (2) predefined functions that can be integrated within the bitstream parser FU during the parser instantiation process. The `rvc:ext` type can be only applied to an `xsd:element` element. An example of Variable Length Decoding is provided below:

```
<xsd:element name="DCTCoefficient" type="rvc:ext" rvc:extName="VLD"
rvc:extParams="MV_START_INDEX"/>
```


For external Functional Unit:

A communication scheme (described in I.3) is set up to make the link with this external Functional Unit. The `rvc:extName` and `rvc:extParams` attributes help in making this link by specifying the name of the ports used to connect the parser and the Functional Unit.

Connections with an external FU are necessary to decode the DCT coefficients, which are Variable Length Codes. These coefficients shall be decoded using ISO/IEC 14496-2:2004, Table B.16 (the VLC table). Thus, a connection is established between the parser and the corresponding Functional Unit to decode this element of syntax. Example of such a communication protocol is detailed in I.3.1.

For parser plug-in function:

In the case of the bitstream syntax element decoded by a predefined function in the to-be-instantiated bitstream parser FU, `rvc:ext` can be used to designate the function and to provide necessary parameters to the function. The name of plug-in function should be defined in the `rvc:extName` attribute and `rvc:extParams` attribute should be used to deliver necessary parameters to the function. The detailed interaction mechanism is described in I.3.2.

C.4.3.3 xsd:group element

The `xsd:group` element is used to define a set of elements of syntax. This element allows having a hierarchical bitstream description. In a BSDL schema, there are several ways of accessing different levels of hierarchy in the bitstream. However, in RVC-BSDL, only the `xsd:group` element shall be used to express different levels of hierarchy into the bitstream.

Syntax:

When declaring an `xsd:group`:

```
<xsd:group
     name="string"
>
Children: {xsd:sequence, xsd:choice}
</xsd:group>
```

When calling an `xsd:group`:

```
<xsd:group
     ref="string"
     bs2:if="Expression"
     bs2:ifNext="NumericLiteral(, NumericLiteral)"
     bs2:nOccurs="(Expression | unbounded)"
     rvc:iterator="Expression"
     rvc:iteratorInit="Expression"
>
Children: none
</xsd:group>
```

Semantics:

The `xsd:group` element can be used in two different cases, the declaration case and the calling case.

An `xsd:group` element that appears directly under the `xsd:schema` element is considered as a group declaration. A group may represent a specific bitstream hierarchy and may contain several syntax elements.

An `xsd:group` element under the `xsd:sequence` or `xsd:choice` element is considered as a group calling. The `ref` attribute is used to designate the name of the `xsd:group` to be called.

The `xsd:group` element can be used hierarchically, that is, a called `xsd:group` element may call other `xsd:group` element again.

When calling a group, conditional and loop statements can be used. For attributes describing conditional statements, see C.4.4.1 and C.4.4.2 for their syntax and semantics.

Example:

The example below shows how to use the `xsd:group` element. In the bitstream, when the parser meets this element:

```
<xsd:group ref="GroupOfVideoObjectPlane"/>
```

The parser refers to the definition of the group, which is:

```
<xsd:group name="GroupOfVideoObjectPlane">
 <xsd:sequence>
   <xsd:element name="group_of_vop_start_code" type="bs1:b32"/>
   <xsd:element name="time_code" type="bs1:b18"/>
   <xsd:element name="closed_gov" type="bs1:b1"/>
   <xsd:element name="broken_link" type="bs1:b1"/>
   <xsd:element name="next_start_code" type="bs1:align8"/>
   <xsd:group ref="user_data" bs2:ifNext="1B2"/>
 </xsd:sequence>
</xsd:group>
```

The above example shows a way to express a hierarchy in the bitstream. The `xsd:group` element can be used hierarchically.

Example:

The example below shows how to use the `xsd:group` element. During the bitstream parsing process, when the parser meets this element:

```
<xsd:group ref="GroupOfVideoObjectPlane"/>
```

The parser refers to the definition of the group, which is:

```
<xsd:group name="GroupOfVideoObjectPlane">
 <xsd:sequence>
   <xsd:element name="group_of_vop_start_code" type="bs1:b32"/>
   <xsd:element name="time_code" type="bs1:b18"/>
   <xsd:element name="closed_gov" type="bs1:b1"/>
   <xsd:element name="broken_link" type="bs1:b1"/>
   <xsd:element name="next_start_code" type="bs1:align8"/>
   <xsd:group ref="user_data" bs2:ifNext="1B2"/>
 </xsd:sequence>
</xsd:group>
```

The above example shows a way to express a hierarchy in the bitstream.

C.4.3.4 xsd:sequence element

The `xsd:sequence` element constructs a block of sequentially arranged syntax elements.

NOTE The `xsd:sequence` element can be used to implement if, for, or while statement on BSD.

Syntax:

```
<xsd:sequence
     bs2:if="Expression"
     bs2:ifNext="NumericLiteral(, NumericLiteral)"
     bs2:nOccurs="(Expression | unbounded)"
     rvc:iterator="Expression"
     rvc:iteratorInit="Expression"
>
Children: {xsd:sequence, xsd:choice, xsd:group, xsd:element}
</xsd:sequence>
```

Semantics:

The `xsd:sequence` element can be used hierarchically within the scope of other `xsd:sequence` element to gather a list of consecutive elements of syntax which have conditions in common.

BSDL attributes for conditional statements, such as `bs2:if` or `bs2:nOccurs`, can be used to apply a specific condition to a block of elements. For attributes describing conditional statements, see C.4.4.1 and C.4.4.2 for their syntax and semantics.

Example:

The elements `requested_upstream_message_type` and `newpred_segment_type` exist only if the variable `newpred_enable` equals to "1".

```
<xsd:sequence bs2:if="newpred_enable=1">
   <xsd:element name="requested_upstream_message_type" type="bs1:b2"/>
   <xsd:element name="newpred_segment_type" type="bs1:b1"/>
</xsd:sequence>
```

C.4.3.5 xsd:choice element

The `xsd:choice` element is used to make a choice between two or several elements of syntax.

NOTE The `xsd:choice` element can be used to implement if-else or switch-case structure on BSD.

Syntax:

```
<xsd:choice
     bs2:if="Expression"
     bs2:ifNext="NumericLiteral(, NumericLiteral)"
     bs2:nOccurs="(Expression | unbounded)"
     rvc:iterator="Expression"
     rvc:iteratorInit="Expression"
>
Children: {xsd:sequence, xsd:choice, xsd:group, xsd:element}
</xsd:choice>
```

Semantics:

The children elements of `xsd:choice` should have a `bs2:if` or `bs2:ifNext` attribute in order to be able to decide which element shall be chosen. The condition on each element shall be defined such as only one choice shall be possible. The evaluation of conditions of the children elements shall be done in consecutive order. Once a child element is chosen, condition evaluation for the remaining children elements will be ignored.

A child element without conditional statement means that the condition is always true; in other words, the child element will always be chosen when no other child element is chosen before. Therefore, an `xsd:group` or an `xsd:element` without condition can be used like an "else" statement in the if-else structure or "default" case in switch-case structure.

Example:

This example describes a simple if-else structure. The first child element of `xsd:choice, next_sc`, is only parsed when `vop_coded` equals to 0. On the other hand, the second child element, `VOPdata` group, is only called when the above element is not processed.

```
<xsd:choice>
   <xsd:element name="next_sc" type="bs1:align8" bs2:if="vop_coded=0"/>
   <xsd:group ref="VOPData"/>
</xsd:choice>
```

C.4.3.6 bs2:variable element

The data parsed from the bitstream may need to be stored in the internal variable managed by the syntax parser FU in order to control the further parsing process or to perform port output behaviour. The `bs2:variable` element allows access to the internal variables within during the syntax parsing process. While the `xsd:element` element with `bs2:partContext` attribute stores data read from the bitstream, the `bs2:variable` element can assign arbitrary value into variable using mathematical or logical expressions.

Syntax:

```
<bs2:variable
     name="string"
     value="Expression{, Expression}"
     bs2:position="Expression{, Expression})"
     rvc:port="string"
     bs2:if="Expression"
>
Children: none
</bs2:variable>
```

Semantics:

In RVC-BSDL, the use of `name`, `value` and `rvc:port` attributes under the `bs2:variable` element is optional. The combination of these attributes should be translated to the various bitstream parser actions as follows:

— `name + value`: The value defined in the `value` attribute is assigned to the internal memory which name is defined in the `name` attribute. If the variable is not yet defined, try to define it.

— `name + port`: The value in the memory which name is defined in the `name` attribute is exported through the output port of the syntax parser FU which name is designated in `rvc:port` attribute.

— `value + port`: The value defined in `value` attribute is exported through the output port of the parser FU designated in `rvc:port` attribute. The value will not be saved in the internal memory in this case.

— `name + value + port`: The value defined in `value` attribute is assigned to the memory which name is defined in `name` attribute, and then is exported through the output port of the parser FU designated in `rvc:port` attribute.

If the `bs2:position` attribute is defined, the variable should be considered as an array-type. The array index is defined by the value of the `bs2:position` attribute. Access to a multidimensional array is allowed by describing more than one index separated by a comma (","). Also, batch assignment into

an array-typed variable can be described by declaring more than one expression separated by a comma (",") in the `value` attribute. See C.4.4.3 for the syntax of the `bs2:position` attribute.

Conditional variable assignment can be described by adding the `bs2:if` attribute. See C.4.4.1 for its syntax and semantics.

Example:

The following use case of a `bs2:variable` element defines a new memory location, `mb_type`, and store a value that is derived from the element of a syntax being decoded.

```
<xsd:element name="mcbpc" type="rvc:ext"
rvc:extName="Algo_VLDtableB7_MPEG4part2">
   <xsd:annotation><xsd:appinfo>
       <bs2:variable name="mb_type" value="bitand(last(),7)"/>
   </xsd:appinfo></xsd:annotation>
</xsd:element>
...
<xsd:group ref="motion_vector" bs2:nOccurs="4" bs2:if="mb_type=2"/>
```

In the following case, an array-typed variable, `sps_sl`, is being updated.

```
<bs2:variable name="sps_sl" bs2:position="sps_id, sps_size_id, sps_matrix_id, k"
value="sps_sl_dc[sps_id][sps_size_id][sps_matrix_id-delta][k]"/>
```

In the following example, an array-typed variable, `default_scaling_list_inter`, is declared by batch assignment for array.

```
<bs2:variable name="default_scaling_list_inter" value="16, 16, 16, 16, 17, ..., 71, 91"/>
// default_scaling_list_inter[4]=17
```

C.4.3.7 xsd:simpleType element

This element is used to define a new type of `xsd:element` element.

Syntax:

```
<xsd:simpleType
     name="string"
>
Children: (xsd:union | xsd:restriction)
</xsd:simpleType>
```

Semantics:

The cases in which a new type shall be defined are when

— the type of the current element is conditioned by a variable assigned during the parsing process; in this case, the `xsd:union` child element is used,

— the length in bits of the current element is defined by a variable assigned during the parsing process; in this case, the children elements `xsd:restriction` and (`xsd:length` or `xsd:bitlength`) are used, and

— testing the bitstream through look-ahead parsing with an `xsd:startcode` element (see C.4.3.11).

Example:

To see different examples of definition of a new type, refer to C.4.3.9 to C.4.3.12.

C.4.3.8 xsd:annotation and xsd:appinfo element

BSDL-2 and RVC-BSDL introduces a set of new XML elements to specify the bitstream parsing process in detail. Since XML schema does not allow a user to add his own facets, such new elements shall be used through the annotation mechanism of the XML schema. Therefore, the new elements shall be described as children elements of the `xsd:annotation/xsd:appinfo` combination.

Syntax:

The syntax of an `xsd:annotation` element is:

```
<xsd:annotation>
     Children: xsd:appinfo
</xsd:annotation>
```

The syntax of an `xsd:appinfo` element when it is used within the scope of an `xsd:schema` element is:

```
<xsd:appinfo>
     Children: {rvc:allocation}
</xsd:appinfo>
```

The syntax of an `xsd:appinfo` element when it is used within the scope of an `xsd:element` element is:

```
<xsd:appinfo>
     Children: {bs2:variable}
</xsd:appinfo>
```

The syntax of an `xsd:appinfo` element when it is used within the scope of an `xsd:union` element is:

```
<xsd:appinfo>
     Children: {bs2:ifUnion}
</xsd:appinfo>
```

The syntax of an `xsd:appinfo` element when it is used within the scope of an `xsd:restriction` element is:

```
<xsd:appinfo>
     Children: (bs2:bitLength | bs2:startcode)
</xsd:appinfo>
```

Semantics:

According to the parent element in which this element is called, there are several possibilities in the semantics of `xsd:annotation/xsd:appinfo` combination.

— If the parent element is an `xsd:schema` element, the annotation mechanism is used for the declaration and initialization of internal variables in the parser FU. See C.4.3.13.

— If the parent element is an `xsd:element` element, the `bs2:variable` element can be used to save variables. See C.4.3.6.

— If the parent element is an `xsd:simpleType` element that is defined with `xsd:union`, the `bs2:ifUnion` elements can be used to define a new user-defined type. See C.4.3.12.

— If the parent element is an `xsd:simpleType` element that is defined with `xsd:restriction`, one of the following elements can be used to define a new user-defined type: `bs2:bitLength` or `bs2:startcode`. See C.4.3.9 to C.4.3.12.

Example:

The example usage of `xsd:annotation` and `xsd:appinfo` is found in C.4.3.2, C.4.3.6, C.4.3.10, C.4.3.11, C.4.3.12, C.4.4.1, and C.4.4.2.

C.4.3.9 xsd:restriction element

This element is used to specify data accuracy. The actual child element of `xsd:restriction` are `bs2:bitLength` or `bs2:startcode`.

Syntax:

```
<xsd:restriction
     base="(bs1:b32 | bs1:byteRange)"
>
Children: xsd:annotation
</xsd:restriction>
```

Semantics:

The base type should correspond with the data type restriction method declared within the `xsd:restriction` element. The base type shall be `bs1:b32` when the child element is `bs2:bitLength`, while the `bs1:byteRange` data type should be used when the child element is a `bs2:startcode` element.

Example:

The usage of an `xsd:restriction` element is shown in C.4.3.9 and C.4.3.10.

C.4.3.10 bs2:bitLength element

This element specifies the size in bits of the current element, which has been defined as a new type using the `xsd:simpleType` construct.

Syntax:

```
<bs2:bitLength
     value="Expression"
>
Children: none
</bs2:bitLength>
```

Semantics:

The value attribute defines the length of the syntax element. The size in bits of the current element can be stored in a variable, which has been assigned during the parsing process.

Example:

The `VOPTimeIncrementType` type instantiates elements of size defined in the variable `vopTimeIncrementBits`.

```
<xsd:simpleType name="VOPTimeIncrementType">
 <xsd:restriction base="bs1:b32">
   <xsd:annotation><xsd:appinfo>
     <bs2:bitLength value="vopTimeIncrementBits"/>
   </xsd:appinfo></xsd:annotation>
 </xsd:restriction>
</xsd:simpleType>
```

This type can be used by an `xsd:element` declaration as the following example:

```
<xsd:element name="VOPTimeIncrement" type="VOPTimeIncrementType"/>
```

C.4.3.11 bs2:startCode element

In some coding formats, a data segment is read until a start code is found. A start code consists of one or more byte-sequences that indicate the start of a new data segment. For example in ISO/IEC 1449-10 (Advanced Video Coding), the content of a NALUnit is read until the start code `0x00000001` is seen (which indicates the start of the subsequent NAL Unit). The `bs2:startCode` element is used to process such a data segment.

Syntax:

```
<bs2:startCode
     value="HexadecimalValue">
Children: none
</bs2:startCode>
```

Semantics:

The value defined in the value attribute should be interpreted as the starting bits of the next syntax element. Look-ahead parsing should be used to test the following bits.

The `bs1:byteRange` data type is only allowed when it is used with the `bs2:startCode` element.

Example:

The type `rbspType` allows bitstream parser FU to parse bitstream continuously until it finds a new start code starting with `"00000001"` bit. The following bitstream syntax element could be a 32-bit start code since the bitstream reading pointer will not be moved when the bits defined in `bs2:startCode` element is tested.

```
<xsd:simpleType name="rbspType">
 <xsd:restriction base="bs1:byteRange">
   <xsd:annotation><xsd:appinfo>
     <bs2:startCode value="00000001"/>
   </xsd:appinfo></xsd:annotation>
 </xsd:restriction>
</xsd:simpleType>
```

C.4.3.12 xsd:union element and bs2:ifUnion element

The combination of these elements allows users to choose the type of an element among a list of member types according to some conditions defined in the `bs2:ifUnion` element.

Syntax:

The syntax of an `xsd:union` element is:

```
<xsd:union
     memberTypes="{bs1:b1 - bs1:b32 | bs1:align8 | user-defined type}"
>
Children: xsd:annotation
</xsd:union>
```

The syntax of a `bs2:ifUnion` element is:

```
<bs2:ifUnion
value="Expression"
>
Children: none
</bs2:ifUnion>
```

Semantics:

The `bs2:ifUnion` element specifies the conditions under which the corresponding type is chosen. The number of `bs2:ifUnion` elements that shall appear is equal to the number of member types defined in the above `xsd:union` element.

Example:

The type `SpriteType` instantiates elements of type `bs1:b1` or `bs1:b2`. The type `bs1:b1` is chosen if the condition `"volVersion=1"` is `"true"`. The type `bs1:b2` is chosen if the condition `"volVersion=1"` is `"false"`.

```
<xsd:simpleType name="SpriteType">
 <xsd:union memberTypes="bs1:b1 bs1:b2">
   <xsd:annotation><xsd:appinfo>
     <bs2:ifUnion value="volVersion=1"/>
     <bs2:ifUnion value="volVersion != 1"/>
   </xsd:appinfo></xsd:annotation>
 </xsd:union>
</xsd:simpleType>
```

C.4.3.13 rvc:allocation element

The `rvc:allocation` element in RVC-BSDL allows description of the list of internal variables which are necessary in controlling the bitstream parsing process. Information described in this element can be used by the parser instantiation mechanism or the bitstream parser itself to estimate the size of variable memory. Especially, this element is useful when using array-typed variables of multiple dimensions.

Syntax:

```
<rvc:allocation
     name="Expression"
     rvc:isArray="(true | false)"
     rvc:dimension="NumericLiteral {, NumericLiteral}"
     value="Expression{, Expression}">
Children: none
</rvc:allocation>
```

Semantics:

A single `rvc:allocation` element represents a single internal variable. The `name` attribute defines the name of the variable, which can be used as identifier in the `xsd:element` and `bs2:variable` element. The value of the `name` attribute should be unique among all `rvc:allocation` declarations.

The `rvc:isArray` attribute defines whether the given variable is an array-type one or not. The variable should be considered as an array when this attribute is set to `"true"`.

The `rvc:dimension` attribute defines the dimension and the depth per dimension for the variable. More than one depth, separated by a comma (","), can be described to declare a multidimensional array variable.

The `value` attribute can be used to define initial value(s) to the allocated variable. The syntax is the same with that of the `bs2:variable` element.

Example:

The example below declares an array-typed internal variable, which is four-dimensional.

```
<rvc:allocation name="sps_sl" rvc:isArray="true"
rvc:dimension="15,4,6,64"/>
```

C.4.4 Syntax of the attributes

This subclause describes the syntax of the attributes used with the schema elements.

NOTE The syntax of the attributes that are derived from the existing standards (e.g. XML schema or BSDL) is as described in the respective standards. This subclause only describes new functionalities on the existing attributes and new attributes defined by RVC-BSDL.

C.4.4.1 bs2:if attribute

The `bs2:if` attribute specifies a conditional test to determine a syntax element is to be parsed or not. Different from BSDL-2, the `bs2:if` attribute in RVC-BSDL schema is described in a simplified expression syntax, which is defined in C.4.5. Also, it should be noted that the `bs2:if` attribute is newly allowed for the `bs2:variable` element in RVC-BSDL.

Syntax:

This attribute is allowed for the following elements:

— syntax element blocks (`xsd:sequence` and `xsd:choice`);

— bitstream syntax elements (`xsd:element`, `xsd:group` of the group calling case);

— variable manipulation (`bs2:variable`).

Semantics:

If the `bs2:if` attribute is defined for an `xsd:element` element that contains `bs2:variable` elements as children, all contained `bs2:variable` elements are also affected by the `bs2:if` condition of their parent element. Meanwhile, if the `bs2:if` attribute is defined for a single `bs2:variable` element, the conditional test only affects the single element.

Example:

The following example shows an `xsd:element` that contains several `bs2:variable` calculations. If the `bs2:if` condition of an `xsd:element` element (`"btype_isQ==1"`) is evaluated as `"false"`, the two-bit-length bitstream syntax element (`"dquant"`) is not read from the bitstream, and all `bs2:variable` elements contained within the element is also negated. On the other hand, if the `bs2:`

condition of bs2:variable element (e.g. "dquant==0") is evaluated as "false", only a single bs2:variable element is negated.

```
<xsd:element name="dquant" type="bs1:b2" bs2:if="btype_isQ == 1" bs2:partContext="true">
<xsd:annotation><xsd:appinfo>
    <bs2:variable name="vop_quant" value="vop_quant-1" bs2:if="dquant==0"/>
    <bs2:variable name="vop_quant" value="vop_quant-2" bs2:if="dquant==1"/>
    <bs2:variable name="vop_quant" value="vop_quant+1" bs2:if="dquant==2"/>
    <bs2:variable name="vop_quant" value="vop_quant+2" bs2:if="dquant==3"/>
    <bs2:variable name="vop_quant" value="31" bs2:if="vop_quant> 31"/>
    <bs2:variable name="vop_quant" value="1" bs2:if="1> vop_quant"/>
</xsd:appinfo></xsd:annotation>
</xsd:element>
```

C.4.4.2 bs2:nOccurs, rvc:iterator and rvc:iteratorInit attribute

The bs2:nOccurs attribute specifies the number of occurrences of a single syntax element or a block of syntax elements. The value of this attribute may be a number or an expression. In either case, the evaluated number signifies the number of repetition. Different from BSDL-2, a bs2:if attribute in RVC-BSDL schema is described in a simplified expression syntax, which is defined in C.4.5. Also, it should be noted that a bs2:nOccurs attribute is newly allowed for the bs2:variable element in RVC-BSDL.

Syntax:

This attribute is allowed for the following elements:

— syntax element blocks (xsd:sequence and xsd:choice);

— bitstream syntax elements (xsd:element, xsd:group of the group calling case);

— variable manipulation (bs2:variable).

Semantics:

In RVC-BSDL, when the value of a bs2:nOccurs attribute is set to "unbounded", this should be interpreted as an infinite loop. In this case, a bs2:if attribute can be used to define the break condition for the loop. The combination of "bs2:nOccurs="unbounded"" and a bs2:if attribute should be interpreted that this syntax element (or this block of elements) should repeatedly be parsed while the expression given in the bs2:if attribute is evaluated as "true". The condition described in bs2:if should be tested before each repetition, like a typical "while" statement.

The rvc:iterator attribute can be used along with a bs2:nOccurs attribute to designate specific internal variable as an incremental iterator. The initial value of the iterator variable can be set by an rvc:iteratorInit attribute; otherwise, the initial value is zero (0) by default. The iterator variable will be increased by one for every loop evoked by a bs2:nOccurs attribute. The loop will be stopped when the value iterator variable equals to the value assigned in a bs2:nOccurs attribute.

Example:

The following example shows a usage of `bs2:nOccurs` with `rvc:iterator`. The iterator variable (`"i"`) contributes to the control flow within the conditional loop defined using `xsd:sequence`.

```
<xsd:sequence rvc:iterator="i" bs2:nOccurs="vps_num_hrd_parameters">
<xsd:element name="hrd_layer_set_idx" type="rvc:ext"
rvc:extName="EF_EXP_GOLOMB_U"/>
<xsd:element name="cprms_present_flag" bs2:if="i == 0"/>
</xsd:sequence>
```

The following example shows more complicated control flow that exploits both `rvc:iterator` and `rvc:iteratorInit` attributes.

```
<xsd:sequence rvc:iterator="i"
rvc:iteratorInit="pcRPS[STRPS_sps_id][STRPS_idx][0]"
bs2:nOccurs="pcRPS[STRPS_sps_id][STRPS_idx][2]">
<xsd:element name="delta_poc_s1_minus1" type="rvc:ext"
   rvc:extName="EF_EXP_GOLOMB_U" bs2:partContext="true">
<xsd:annotation><xsd:appinfo>
<bs2:variable name="t1" value="3+i"/>
<bs2:variable name="prev" value="delta_poc_s1_minus1-1"/>
<bs2:variable name="pcRPS" bs2:position="STRPS_sps_id, STRPS_idx, t1"
             value="prev"/>
   </xsd:appinfo></xsd:annotation>
</xsd:element>
<xsd:element name="used_by_curr_pic_s1_flag" type="bs1:b1"
   bs2:partContext="true">
   <xsd:annotation><xsd:appinfo>
      <bs2:variable name="t1" value="67+i"/>
      <bs2:variable name="pcRPS" bs2:position="STRPS_sps_id, STRPS_idx, t1"
             value="used_by_curr_pic_s1_flag"/>
   </xsd:appinfo></xsd:annotation>
</xsd:element>
</xsd:sequence>
```

C.4.4.3 bs2:position attribute

The `bs2:position` attribute specifies access to array-typed internal variable (i.e. `"node-set"` in BSDL).

Syntax:

This attribute is allowed for an `xsd:element` element and a `bs2:variable` element.

Semantics:

The semantics of this attribute is changed from its original form in BSDL-2 as follows:

— The use of the `bs2:position` attribute is newly allowed for the `xsd:element` element to facilitate data management during the bitstream parsing process. See C.4.3.2 for more information.

— The value of a `bs2:position` attribute should be described in a simplified expression syntax, which is defined in C.4.5.

— Notation of multiple dimension array index is allowed. Each array index should be separated by comma (",") with each other.

Example:

The usage of the `bs2:position` attribute can be found in C.4.3.2 (`xsd:element`) and C.4.3.6 (`bs2:variable`).

C.4.4.4 rvc:extName and rvc:extParams attribute

These attributes describe the communication scheme with external functions or other FUs during the bitstream parsing process.

Syntax:

These attributes are only allowed for the `xsd:element` element with a `"rvc:ext"` type.

Semantics:

The `rvc:extName` and `rvc:extParams` attributes are used to describe the bitstream syntax elements that should be parsed by externally defined algorithms. These elements only can be used in an `xsd:element` element which type is set to `"rvc:ext"`, and the definition of a `rvc:extName` attribute is compulsory in this case.

The `rvc:extName` attribute indicates the unique name or identifier of an external algorithm. The name may represent the name of an externally defined FU or the identifier of plug-in parsing function within the bitstream parser FU. The `rvc:extParams` attribute can be used to define parameters for the external algorithm. Multiple parameters should be separated with a comma (",").

NOTE See I.3 for the further details on the communication scheme between the syntax parser FU and the external algorithms.

Example:

An example of these attributes can be found in C.4.3.2.1 (`rvc:ext`).

C.4.4.5 rvc:port attribute

This attribute specifies the token output from the syntax parser FU.

Syntax:

This attribute is allowed for an `xsd:element` element and a `bs2:variable` element.

Semantics:

The parsers built from RVC decoder configurations generate data tokens on different output ports. Consequently, a mechanism specifying the correspondence between the tokens, corresponding to the different elements of syntax and the output ports on which they have to be sent as output tokens, is necessary to fully specify a decoder configuration. A special attribute has been added in order to define the port on which the data is sent. The `rvc:port` attribute is used to indicate that the corresponding element of syntax shall be available outside the parser for further processing operated by the network of FUs. This attribute is applied to `xsd:element` (C.4.3.2) and `bs2:variable` (C.4.3.6).

Example:

```
<xsd:element name="video_object_layer_width" type="bs1:b13"
rvc:port="width"/>
```

Thus, the element "video_object_layer_width" is available as a token on the port "width" of the parser. Obviously, the connections of the parser to the network of FUs are reported in the description of the RVC decoder configuration connected to the port "width". It is available in the specification of the FU Network Description (FND), which is given as an input of the whole framework (see Figure 2). In the above example, the corresponding FND shall contain the description of a link connecting the output port "width" of the parser and an input port of an FU.

C.4.4.6 rvc:rootGroup **attribute**

This attribute is only allowed for xsd:schema element: see C.4.3.1 for its syntax and usage.

C.4.4.7 rvc:isArray **and** rvc:dimension **attribute**

These attributes are only allowed for rvc:allocation element: see C.4.3.13 for their syntax and usage.

C.4.5 Syntax of the expressions

This subclause describes the syntax of the expressions used in the attributes.

Expression → PrimaryExpression
| UnaryOperator PrimaryExpression
| PrimaryExpression Operator PrimaryExpression

PrimaryExpression → 'max('Expression','Expression')'
| 'min('Expression','Expression')'
| 'numbits('Expression')'
| 'bitand('Expression','Expression')'
| 'bitor('Expression','Expression')'
| 'bitnot('Expression')'
| 'rshift('Expression','Expression')'
| 'lshift('Expression','Expression')'
| 'log2('Expression')'
| 'last()'
| ExpressionLiteral

ExpressionLiteral → NumericLiteral | VariableExpression | true | false

VariableExpression → VariableName{ArrayIndex}

ArrayIndex → '['Expression']'{ArrayIndex}

UnaryOperator → ('not')

Operator → ('=' | 'lt' | 'lte' | '>' | '>= ' | '!=' | 'and' | 'or' | '*' | '/' | '+' | '-' | '^' | 'mod')

The VariableName is the name of an element that appears on the current BSD or the name of a variable defined by a bs2:variable element. Accordingly, it should be a qualified name for an XML element. RVC-BSDL does not use an XPath model to designate a specific element or variable. All elements and variables are considered global in the parser FU and, therefore, are accessible from any expression syntax without a path designation. See the following example:

```
<xsd:sequence bs2:if="video_object_layer_shape != 2 and vop_coding_type=1">
 <xsd:element name="vop_rounding_type" type="bs1:b1"/>
</xsd:sequence>
```


A variable can be considered as an array type if it is followed by a square bracket.

```
<bs2:variable name="CBP[4]" value="bitand(rshift(cbpc,4),1)"/>
```

Multiple dimensions are allowed for the array notation.

```
<bs2:variable name="pcRPS" bs2:position="sps_id, t1, i"
value="pcRPS[sps_id][short_term_ref_pic_set_idx][i]"/>
```

Additionally, less-than ("<") and less-than-or-equal-to ("<=") operators are not available in RVC-BSDL because the use of the less-than symbol ("<") within an attribute value is not allowed in the XML grammar. When describing mathematical expressions, these operators should be replaced with literals ("lt" and "lte") or bypassed by exchanging the order of operands. (e.g. from "x<5" to "5>x").

C.4.5.1 last() function

The `last()` function is a predefined function that returns the value of the latest bitstream syntax element parsed, regardless of whether the syntax element is marked with `bs2:partContext` or not. The following example shows a use case of the `last()` function.

```
<xsd:element name="vop_time_increment_resolution" type="bs1:b16">
<xsd:annotation>
   <xsd:appinfo>
      <bs2:variable name="vopTimeIncrementBits" value="numbits(last())"/>
   </xsd:appinfo>
</xsd:annotation>
</xsd:element>
```

In this example, the `bs2:variable` can use the value read by the `xsd:element` "vop_time_increment_resoltion" even though the element is not specified as a named variable by `bs2:partContext="true"`.

C.4.5.2 log2() function

The `log2()` function returns the base 2 logarithm of the given value. The decimal points are truncated. This function can be used to calculate the minimum number of bits to represent the given value in unsigned binary integer (i.e. as a "numbits" function). For instance, `log2(14)` returns 4, as 14_{10} is 1110_2.

C.4.6 Syntax of the data types

This subclause describes the syntax of the data types.

NumericLiteral → IntegerLiteral | '0x'HexadecimalValue
IntegerLiteral → IntegerDigit{IntegerDigit}
IntegerDigit → ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')
HexadecimalValue → HexadecimalDigit{HexadecimalDigit}
HexadecimalDigit → ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F')

C.5 Connections between the syntax parser and the FU network

The Syntax Parser and the network of FU shall be connected together. Thus, a communication scheme between the syntax parser and Functional Unit is necessary.

C.5.1 General output ports

For each distinguishable names designated for the `rvc:port` attribute in the RVC-BSDL description, output ports shall be created for the syntax parser FU. The following code creates an output port named `"ACPRED"`.

```
<xsd:element name="AC_pred_flag" type="bs1:b1" bs2:partContext="true" rvc:port="ACPRED"/>
```

C.5.2 Output ports with feedback ports

The following code shows an example of BSD, illustrating the connection of the Syntax parser to an FU with feedback ports generated by the element with `rvc:ext datatype`.

```
<xsd:element name="mcbpc" type="rvc:ext" rvc:extName="EF_VLD"
rvc:extParams="16" bs2:partContext="true"/>
```

The element name `"mcbpc"` is decoded by an external algorithm, which is indicated by the data type `rvc:ext`. The name of the external algorithm, `"EF_VLD"`, is designated by the `rvc:extName` attribute. When the external algorithm `"EF_VLD"` is not available as a plug-in function within the instantiated parser FU, communication should be established between the syntax parser FU and an external FU.

Whenever a connection to a Functional Unit is established, the induced ports of the parser are:

— Bitstream output port: an output port which name is value of the `rvc:extName` attribute, e.g. `mcbpc`. This port is always created. It is used to send the bitstream to be parsed to the FU.

— Parameter output port: an output port which name is value of the `rvc:extParams` attribute, e.g. `16`. This port is created only when the `rvc:extParams` attribute is set. It is used to send the necessary parameters to the FU.

— Status feedback port: an input port which name is value of the `rvc:port` attribute followed by the suffix `"_f"` (e.g. `algo_mv_f`). This port is always created. It is used to acknowledge the status of the FU each time the parser sends data.

— Value feedback port: an input port which name is value of the `rvc:port` attribute followed by the suffix `"_data"` (e.g. `algo_mv_data`). This port is created only when the attribute `bs2:partContext` is set to `"true"` in the current element. It is used to return the decoded value to the parser, which can use this value to continue its parsing process.

In order to know if the parser can go to the next element of syntax or not, a communication protocol between the syntax parser and the FU has been defined.

a) The parser sends data on the port `EF_VLD` and parameters on the port `EF_VLD_p`.

b) The FU receives the data and warns the parser (though the `EF_VLD_f` port):

   1) if it needs more data (value of the data to return to the parser=false), go to a), or

   2) if it has finished (value of the data to return to the parser=true), go to c).

c) The value received via the input port `EF_VLD_data` is set as the value of the syntax element in the parser.

d) The parser can continue parsing the other elements of syntax.

Note that the value returned through the value feedback port can be sent through a designated output port if the element has `rvc:port` attribute. The ports used for the connection with the bitstream parsing FUs are separated from the general output ports described in C.5.1.

The examples of VLD decoding process using such communication scheme are shown in Annex I.

Annex D
(normative)

Specification of the RVC-CAL language

D.1 Generalities

The CAL language is a dataflow-oriented language that was developed as a subproject of the Ptolemy Project at the University of California, Berkeley. The final CAL language specification was released in December 2003. The specification provided in this annex is the subset of the CAL language called RVC-CAL used in the MPEG RVC framework. The subset has been defined so as to keep all data types and operators that are necessary in the RVC framework scope, excluding data types and operators that cannot be easily converted to software or hardware implementations.

RVC-CAL is a textual language that is used to define the functionality of dataflow components originally called "actors", which in the MPEG RVC framework are the FUs composing the RVC video tool library. FUs can then be configured into decoders using an XML-based specification language (the RVC FU language called FNL). Therefore, to build the RVC framework, two normative elements are necessary:

— RVC-CAL used to specify the behaviour of the FUs that constitute the RVC video tool library;

— FNL used to specify RVC decoder configurations using FUs from the RVC video tool library.

The XML-based specification of "network of actors" or better in RVC "Configuration of FUs" can be edited and simulated by tools available in the RVC reference software.

It is worth remarking that what in RVC-CAL is called an "actor" exactly corresponds to what in MPEG RVC is called an FU. In fact, an actor is a modular component that encapsulates its own state, no other actor has access to it, and nothing other actors can do to modify the state of an actor. The only interaction between actors is through FIFO channels connecting "output ports" to "input ports," which are used to send and receive "tokens." This strong encapsulation leads to loosely coupled systems with very manageable and controllable actor interfaces. The modularity of an actor assembly fosters concurrent development, it facilitates maintainability and understandability and makes systems constructed this way more robust and easier to modify. All these features correspond to what is required by the MPEG RVC framework.

A "token" is a unit of data (of potentially arbitrary size and complexity) that is sent and received atomically. Each actor input is associated with a queue of tokens waiting in front of it. When a token is sent it is conceptually placed in the queue of each input connected to the output the token originates from. Eventually, the receiving FUs will read it, and thereby consume it, i.e. remove it from the input queue.

Every FU executes in a (possibly unbounded, i.e. non-terminating) sequence of steps, also called "transitions." During each such step, an FU may do any of the following three things:

— read and consume input tokens;

— modify its internal state;

— produce output tokens.

An FU may perform a number of different transitions and can be disabled at any point when it cannot perform any further transitions.

The specification of an FU in RVC-CAL is structured into "actions." Each action defines a kind of transition the FU can perform under some conditions. These conditions may include

— the availability of input tokens,

— the value of input tokens,

— the (internal) state of the FU, and

— the priority of that action (see below).

An FU may contain any number of actions. Its execution follows a simple cycle.

1) Determine, for each action, whether it is enabled, by testing all the conditions specified in that action.

2) If one or more actions are enabled, pick one of them to be fired next.

3) Execute that action, i.e. make the transition defined by it.

4) Go to Step 1.

Steps 1 and 2 are called "action selection." For many complex FUs, such as the parser of an MPEG-4 SP decoder, defining the logic of how an action is chosen is the core of the implementation of the processing in FU form. RVC-CAL provides a number of language constructs for structuring the description of how actions are to be selected for firing. These include:

— action guards: conditions on the values of input tokens and/or the values of FU state variables that need to be true for an action to be enabled;

— finite state machine: the action selection process can be governed by a finite state machine, with the execution of an action causing a transition from one state to the next;

— action priorities: actions may be related to each other by a partial priority order, such that an action will only execute if no higher-priority action can execute.

In this way, the process of action selection is specified in a declarative manner in each RVC FU. As a result, the FU specification becomes more compact and easier to understand.

Once selected, an action is executed. The code describing an action itself is for the most part ordinary imperative code, as can be found in languages such as Pascal, Ada or C, there are loops, branches, assignments, etc. Only the token input/output of an action is specified separately and in a declarative manner.

In other words, the RVC-CAL language provides naturally the appropriate constructs that have been identified by RVC requirement work as essential elements for building the MPEG RVC framework with the capacity of "encapsulating" coding tools functionalities in a very natural manner without needing any particular restriction or specific coding style on the usage of the language construct.

D.2 Overview

D.2.1 General

This clause describes the RVC-CAL, a profile of the CAL actor language to be used by the MPEG Reconfigurable Video Coding Framework.

D.2.2 Actors

The concept of actor as an entity that is composed with other actors to form a concurrent system has a rich and varied history. Some important milestones are found in References [3], [4], [5], [6] and [9]. A formal description of the notion of actor underlying this specification can be found in D.1, which is based on the work in References [10] and [7]. Intuitively, an actor is a description of a computation on

sequences of tokens (atomic pieces of data) that produces other sequences of tokens as a result. It has input port(s) for receiving its input tokens, and it produces its output tokens on its output port(s).[1)]

The computation performed by an actor proceeds as a sequence of atomic steps called rings. Each ring happens in some actor state, consumes a (possibly empty) prefix of each input token sequence, yields a new actor state, and produces a finite token sequence on each output port.

Several actors are usually composed into a network, a graph-like structure (often referred to as a model) in which output ports of actors are connected to input ports of the same or other actors, indicating that tokens produced at those output ports are to be sent to the corresponding input ports. Such actor networks are of course essential to the construction of complex systems, but we will not discuss this subject here, except for the following observations:

— A connection between an output port and an input port can mean different things. It usually indicates that tokens produced by the former are sent to the latter, but there are a variety of ways in which this can happen: token sent to an input port may be queued in FIFO fashion, or new tokens may "overwrite" older ones, or any other conceivable policy. It is important to stress that actors themselves are oblivious to these policies: from an actor point of view, its input ports serve as abstractions of (prefixes of) input sequences of tokens, while its output ports are the destinations of output sequences.

— Furthermore, the connection structure between the ports of actors does not explicitly specify the order in which actors are read. This order (which may be partial, i.e. actors may fire simultaneously), whether it is constructed at runtime or whether it can be computed from the actor network, and if and how it relates to the exchange of tokens among the actors, all these issues are part of the interpretation of the actor network.

The interpretation of a network of actors determines its semantics and it determines the result of the execution, as well as how this result is computed, by regulating the flow of data as well as the flow of control among the actors in the network. There are many possible ways of interpreting a network of actors, and any specific interpretation is called a model of computation, see References [11] and [12]. Actor composition inside the actor model, where CAL is based on, has been studied in Reference [8].

As far as the design of a language for writing actors is concerned, the above definition of an actor and its use in the context of a network of actors suggests that the language should allow making some key aspects of an actor definition explicit. These are, among others:

— the port signature of an actor (its input ports and output port(s), as well as the kind of tokens the actor expects to receive from or be able to send to);

— the code executed during a ring, including possibly alternatives whose choice depends on the presence of tokens (and possibly their values) and/or the current state of the actor;

— the production and consumption of tokens during a ring, which again may be different for the alternative kinds of rings;

— the modification of state depending on the previous state and any input tokens during a ring.

D.2.3 Units

Unlike an actor, a unit does not compute anything. A unit is used to declare "constants", "functions" and "procedures" that can be referenced or imported into an actor. It cannot contain mutable variables which would violate the design constraint that actors do not share state. Units help in factorizing the code in order not to duplicate function declarations or FU constants

1) The notion of actor and firing is based on the one presented in Reference [10], extended by a notion of state in Reference [7].

D.3 Introductory remarks

D.3.1 General

Throughout this document, fragments of RVC-CAL syntax along with (informal) descriptions of what these are supposed to mean will be presented. In order to avoid ambiguity, a few conventions, as well as the fundamental syntactic elements (lexical tokens) of the RVC-CAL language are introduced.

D.3.2 Lexical tokens

RVC-CAL has the following kinds of lexical tokens:

Keywords. Keywords are special strings that are part of the language syntax and are consequently not available as identifiers. See D.12.3 for a list of keywords in RVC-CAL.

Identifiers. Identifiers are any sequence of alphabetic characters of either one of the digits, the underscore character and the dollar sign that is not a keyword. Sequences of characters that are not legal identifiers may be turned into identifiers by delimiting them with backslash characters.

Identifiers containing the $-sign are reserved identifiers. They are intended to be used by tools that generate RVC-CAL program code and need to produce unique names which do not conflict with names chosen by users of the language Consequently, users are discouraged from introducing identifiers that contain the $-sign.

Operators. See D.12.2 for a complete list of RVC-CAL operators.

Delimiters. These are used to indicate the beginning or end of syntactical elements in RVC-CAL. The following characters are used as delimiters: (,), {,}, [,], :.

Comments. Comments are Java-style, i.e. single-line comments starting with "//" and multi-line comments delimited by "/*" and "*/".

Numeric literals. RVC-CAL provides two kinds of numeric literals: those representing an integral number and those representing a decimal fraction. Their syntax is as follows:[2)]

```
                Integer→ DecimalLiteral | HexadecimalLiteral | OctaLiteral
                   Real→ DecimalDigit { DecimalDigit } '.' { DecimalDigit } [ Exponent ]
                         | '.' DecimalDigit { DecimalDigit } [ Exponent ]
                         | DecimalDigit { DecimalDigit } Exponent
        DecimalLiteral→ NonZeroDecimalDigit { DecimalDigit }
    HexadecimalDigit→ '0' ( 'x' | 'X' ) HexadecimalDigit { HexadecimalDigit }
      OctaLiteralDigit→ '0' { OctalDigit }
              Exponent→ ( 'e' | 'E' ) [ '+' | '–' ] DecimalDigit { DecimalDigit ]
  NonZeroDecimalDigit→ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
          DecimalDigit→ '0' | NonZeroDecimalDigit
             OctalDigit→ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
    HexadecimalDigit→ DecimalDigit
                         | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                         | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
```

2) In contrast to all other grammar rules in this document, the following rules do not allow whitespaces between tokens.

D.3.3 Typographic conventions

In syntax rules, keywords are shown in **boldface**, while all other literal symbols are enclosed in single quotes (‘ ‘).

In examples, RVC-CAL code is represented in `monospace`. Semantic entities, such as types, are set in *italic*.

D.3.4 Conventions

We use a form of Backus-Naur form (BNF) to describe the syntax rules. Literal elements are put in quotes (in the case of symbols and delimiters), or set in boldface (in the case of keywords). An optional occurrence of a sequence of symbols *A* is written as [*A*], while any numbers of consecutive occurrences (including none) are written as {*A*}. The alternative occurrence of either *A* or *B* is expressed as *A* | *B*.

We often use plural forms of non-terminal symbols without introducing them explicitly. These are supposed to stand for a comma-separated sequence of at least on instance of the non-terminal; e.g. if A is the non-terminal, we might use As in some production, and we implicitly assume the following definition: As → A { ‘,’ A }.

In the examples reported here, the usual interpretation of expression literals and mathematical operators is assumed, even though strictly speaking these are not part of the language and depend on the environment. A specific implementation of RVC-CAL may not have these operators, or interpret them or the literals in a different manner.

D.3.5 Notational idioms

Like most programming languages, RVC-CAL involves a fair number of syntactical constructs that need to be learned and understood by its users in order to use the language productively. The effort involved in gaining familiarity with the language can be a considerable impediment to its adoption, so it makes sense to employ general guidelines for designing the syntax of constructs, which allow users to make guesses about the syntax if they are unsure about the details of a specific language construction. These guidelines, which define the style of a language, are called notational idioms.

The following is a list of notational idioms guiding the design of RVC-CAL’s language syntax.

Keyword constructs. Many constructs in RVC-CAL are delimited by keywords rather than by more symbolic delimiters. Such constructs are called keyword constructs. Other constructs are delimited by symbols, or are at least partially lacking delimiters (such as assignments, which begin with a variable name; see D.9.2).

Statement head/body separator. Many statements have a similar structure as the one for expressions. For statements, the keywords **do** or **begin** are used as a separator:

```
while n> 0 do k:=f(k); n:=n - 1; end
procedure p (int x) begin
   x:=x + 1;
end
```

D.4 Structure of actor/unit descriptions

D.4.1 Actor description

Each actor description defines a named kind of actor.

Actors are the largest lexical units of specification and translation. Below is the basic structure of an actor:

```
Actor→ ('package' QualifiedName ';')?
        {Import}
        'actor' ID '('ActorPars ')' IOSig ':'
        {VarDecl}
        {Action | InitializationAction}
        [ActionSchedule]
        {PriorityBlock}
        'end'
ActorPar→ Type ID ['=' Expression]
IOSig→ [PortDecls] '==>' [PortDecls]
PortDecl→ {Annotation} Type ID
```

The header of an actor expressed in RVC-CAL contains actor parameters and its port signature. This is followed by the body of the actor, containing a sequence of state variable declarations (D.7.2), actions (D.10), initialization actions (D.10.6), priority blocks (D.11.4) and, at most, one action schedule (D.11.3).

By contrast, actor parameters are values, i.e. concrete objects of a certain type (although, of course, this type may be determined by a type parameter). They are bound to identifiers, which are visible throughout the actor definition. Conceptually, these are non-assignable and immutable, i.e. they may not be assigned to by an actor.

D.4.2 Unit description

A unit can declare functions, procedures, and constants (D.7). A unit can import units. However, a unit cannot import units that lead to a cyclic dependency.

```
Unit:→ ('package' QualifiedName ';')?
        {Import}
        'unit' ID ':'
        (FunDecl | ProcDecl | ConstantVarDecl)*
        'end'
```

D.5 Qualified names and imports

D.5.1 Qualified names

A qualified name is represented with the following rule:

```
QualifiedName: ID('.'ID)*
```

A qualified name with a possible wildcard is allowed only in imports and is defined by:

```
QualifiedNameWithWildCard: QualifiedName '.*'?
```

D.5.2 Declaration of an entity

An entity (actor or unit) may begin with a package directive that declares the package where the unit or actor resides in (similar to Java). In the absence of the package declaration, the unit or actor is considered part of the "default" package, but as in Java this practice is discouraged. The qualified name of an entity is its package followed by a dot and then its identifier. In case the package is not specified, the qualified name is simply the identifier of the entity.

D.5.3 Imports

Qualified names can be imported by imports.

```
Import: 'import' QualifiedNameWithWildCard ';'
```


D.5.4 Reference to unit elements

An actor or unit may reference a variable or function declared in a unit by its qualified name. The qualified name of a variable or function is the name of the variable or function prefixed by the name of the unit it is declared in and a dot, e.g. `MyUnit… myVar`.

An actor or a unit may also import any or all of the variables or functions declared in a unit by using an import statement. Explicit import of one or more variables or functions is done by referencing them by their qualified name, as in

import MyUnit.myVar;

Importing all variables or functions declared by a unit is done by using a wildcard

import MyUnit.*;

Inside an entity, you can use either the qualified name or the simple name of the variable. If you use the simple name of a variable, this variable can be shadowed by another declaration of a variable.

D.6 Data types

D.6.1 Overview

RVC-CAL is fully typed, i.e. it allows and forces programmers to give each newly introduced identifier a type (see D.7.2 for more details on declaring variables).

D.6.2 Variables and types

Each variable or parameter in RVC-CAL may be declared with a variable type. If it is, then this type remains the same for the variable or parameter in the entire scope of the corresponding declaration. Variable types may be related to each other by a subtype relation, $\prec$, which is a partial order on the set of all variable types. When for two variable types t, t' we have $t \prec t'$, then we say that t is a subtype of t', and t' is a supertype of t. Furthermore, t may be used anywhere t' can be used, i.e. variables of subtypes are substitutable for those of supertypes.

It is important that each object has precisely one object type. As a consequence, object types induce an exhaustive partition on the objects, i.e. for any object type t we can uniquely determine the "objects of type t".

IMPLEMENTATION NOTE. Stating that each object has an object type does not imply that this type can be determined at run time, i.e. that there is something like run-time type information associated with each object. In many cases, particularly when efficiency is critical, the type of an object is a compile-time construct whose main use is for establishing the notion of assignability, i.e. for checking whether the result of an expression may legally be stored in a variable. In these scenarios, type information is removed from the runtime representation of data objects.

For each implementation context, it is assumed that there is a set T_V of variable types and T_0 of object types. They are related to each other by an assignability relation, $\leftarrow \subset T_V \times T_0$, which has the following interpretation: For any variable type t_V and object type t_0, $t_V \leftarrow t_0$ if an object of type t_0 is a legal value for a variable of type t_V.

The assignability relation may or may not be related to subtyping, but at a minimum, it shall be compatible with subtyping in the following sense: For any two variable types t_V and t'_V, and any object type t_0:

$$t_V > t'_V \wedge t'_V \leftarrow t_0 \Rightarrow t_V \leftarrow t_0$$

In other words, if an object type is assignable to a variable type, it is also assignable to any of its supertypes.

D.6.3 Type formats

Types are specified as follows:

Type → ID

| ID '('[TypeAttr { ',' TypeAttr }] ')'

TypeAttr → ID ':' Type

| ID '=' Expression

A type that is just an identifier is the name of some non-parametric type or of a parametric type whose parameters take on default values. Examples may be `String`, `int`.

In the next form, the ID refers to a type constructor that has named type attributes which may be bound to either types or values. Type attributes that are bound to types are assigned using the ":" syntax, while values are bound using the "=" syntax.

D.6.4 Predefined types

Required types are the types of objects created as the result of special language constructions, usually expressions. The following are built-in types in RVC-CAL:

— `bool` — the truth values true and false;

— `List(type:T, size=N)` — finite lists of length of N elements with type T;

— `int(size=N)` — signed integers with bit width N;

— `uint(size=N)` — unsigned integers with bit width N;

— `String` — strings of characters;

— `float` — floating point numbers.

D.6.5 Typing rules

D.6.5.1 General

This subclause lists the typing rules for RVC-CAL expressions.

Expression	Type of result
boolean	bool
floating-point number	float
integer with value v	type of int(v)
"xyz"	String
variable var declared with type T	T
unary expression: op e	type of unary(op, e)

Expression	Type of result
binary expression: e1 op e2	type of binary(e1, op, e2)
if cond then e1 else e2 end with cond of type bool	lub(e1, e2)
list[i][j]	type of index(list, i, j)
[e1, e2, …, en: for int i1 in L1 .. H1, for int i2 in L2 .. H2, …, for int in in LN .. HN]	List(type: lub(e1, e2, …, en), size=n * (H1 – L1 + 1) * (H2 – L2 + 1) * … * (HN – LN + 1))

D.6.5.2 Least upper bound (lub)

The least upper bound (lub) of n types is the smallest type that is compatible with the biggest of the given n types. lub(t1, t2, …, tn) is defined as lub(…lub(lub(t1, t2), t3), …, tn).

bool, bool	bool
float, float	float
String, String	String
int(size=S1), int(size=S2)	int(size=max(S1, S2))
uint(size=S1), uint(size=S2)	uint(size=max(S1, S2))
int(size=SI), uint(size=SU) with SI > SU	int(size=SI)
int(size=SI), uint(size=SU) with SU >= SI	int(size=SU + 1)
List(type : T1, size=S1), List(type :T2, size=S2)	List(type:lub(T1, T2), size=max(S1, S2))
any other combinations	invalid

The least upper bound is commutative: lub(t1, t2) is the same as lub(t2, t1).

D.6.5.3 Greatest lower bound (glb)

The greatest lower bound (glb) of n types is the greatest type that is compatible with the smallest of the given n types.

bool, bool	bool
float, float	float
String, String	String
int(size=S1), int(size=S2)	int(size=min(S1, S2))
uint(size=S1), uint(size=S2)	uint(size=min(S1, S2))
int(size=SI), uint(size=SU) with SI > SU	int(size=SU + 1)
int(size=SI), uint(size=SU) with SU >= SI	int(size=SI)
any other combinations	Invalid

NOTE The greatest lower bound has not been defined for List because it is not needed as a typing rule.

D.6.5.4 Type of integer

The size of an integer whose value is v is defined by Formula (D.1):

$$\text{sizeof(v)} = \left\lceil \log2\left(\begin{Bmatrix} \text{v} < 0 \Rightarrow -\text{v} \\ \text{v} \geq 0 \Rightarrow \text{v} + 1 \end{Bmatrix} \right) \right\rceil \tag{D.1}$$

where $\lceil x \rceil$ is ceil(x), which returns the smallest integer that is not less than x.

The type of an integer whose value is v, and size is s=sizeof(v), is defined as:

if v < 0, int(size=s + 1);

if v = 0, uint(size=1);

if v > 0, uint(size=s).

D.6.5.5 Type of unary expressions

Expression	Type of result
bitnot e (or ~e) with e of type T (int or uint)	T
not e with e of type bool	bool
−e with e of type T (int or float)	T
−e with e of type uint(size=s)	int(size=s + 1)
#e with e of type List(type:T, size=S)	S
float_of_int(e) with e of type T (int or uint)	float
int_of_float(e, sz) with e of type float and sz of type int or uint	int(size=sz)
uint_of_float(e,sz) with e of type float and sz of type int or uint	uint(size=sz)

Where float_of_int, int_of_float and uint_of_float are built-ins functions for float to int/uint conversion and vice versa. The conversion to int/uint from float is the truncation conversion towards zero as used in C99 (i.e. int_of_float(5.3, 32) returns 5, and int_of_float(−5.3, 32) returns −5).

D.6.5.6 Type of binary expressions

Expression	Type of result
e1 + e2 with e1 of type String or e2 of type String	String
e1 + e2 with e1 of type List(type:T1, size=S1) e2 of type List(type:T2, size=S2)	List(type:lub(T1, T2), size=S1+S2)
e1 + e2 with e1 of type T1 (int or uint) e2 of type T2 (int or uint)	lub(T1, T2) + 1
e1 − e2 with e1 of type T1 (int or uint) e2 of type T2 (int or uint)	lub(T1, T2) + 1
e1 * e2 with e1 of type int(size=S1) or uint(size=S1) e2 of type int(size=S2) or uint(size=S2)	lub(T1, T2) with size=S1 + S2
e1 << e2 with e1 of type int(size=S1) or uint(size=S1) e2 of type int(size=S2) or uint(size=S2)	S1 + (1 <<S2) − 1
e1 & e2, with e1 of type T1 (int or uint) and e2 of type T2 (int or uint)	glb(T1, T2)
e1 | e2, with e1 of type T1 (int or uint) and e2 of type T2 (int or uint)	lub(T1, T2)
e1 ^ e2 (xor), with e1 of type T1 (int or uint) and e2 of type T2 (int or uint)	lub(T1, T2)
e1 / e2, e1 >> e2, with e1 of type T1 (int or uint) and e2 of type T2 (int or uint)	T1
e1 mod e2, with e1 of type T1 (int or uint) and e2 of type T2 (int or uint)	T2
e1=e2, e1 != e2 with e1 of type T1 and e2 of type T2, if lub(T1, T2) exists	bool

Expression	Type of result
e1 > e2, e1 >= e2, e1 < e2, e1 <= e2, with e1 of type T1 (int or uint or float) and e2 of type T2 (int or uint or float), and if lub(T1, T2) exists	bool
e1 && e2, e1 || e2, with e1 of type bool and e2 of type bool	bool
e1 + e2 with e1 of type float and e2 of type float	float
e1 - e2 with e1 of type float and e2 of type float	float
e1 * e2 with e1 of type float and e2 of type float	float
e1 / e2 with e1 of type float and e2 of type float	float

The type of binary expressions whose operator is +, -, *, /, and where one operand has type float, and the other has type int, uint, or float, is float. In other words, operands with type int or uint are automatically promoted to float.

D.6.5.7 Type of an indexing expression

The type of an indexing expression list[i1][i2]...[in] with a list of type List(type:List(type:...type: List(type:T, size=SN), size=SN_1), ..., size=S1) is T if the type of i1 is not larger than the type of S1 (as obtained with sizeof(S1)), i2 is not larger than sizeof(S2), etc.

If only a subset of indexes is given, say i, then the type of the expression is the type of the i-th inner type.

D.7 Variables, functions and procedures

D.7.1 Overview

Variables are placeholders for values during the execution of an actor. At any given time, they may stand for a specific value, and they are said to be bound to the value that they stand for. The association between a variable and its value is called a binding.

This subclause first explains how variables are declared inside an RVC-CAL source code. It then proceeds to discuss the scoping rules of the language, which govern the visibility of variables and also constrain the kinds of declarations that are legal in RVC-CAL.

D.7.2 Variable declarations

Each variable (with the exception of predefined variables) needs to be explicitly introduced before it can be used. It needs to be declared. A declaration determines the kind of binding associated with the variable it declares, and potentially also it's (variable) type. Following are the following kinds of variable declarations:

— explicit variable declarations (D.7.2),

— actor parameters (D.4),

— input patterns (D.10.2).

The properties of a variable introduced by an explicit variable declaration depend on the form of that declaration.

D.7.2.1 Explicit variable declarations

Syntactically, an explicit variable declaration[3)] is as follows:

3) These declarations are called "explicit" to distinguish them from more "implicit" variable declarations that occur, e.g. in generators or input patterns.

VarDecl → TypeID{‘[‘Expression’]’} [("="|':=’) Expression]

An actor may contain state variable declarations:

StateVarDecl → VarDecl ‘;’

A unit may contain constant variable declarations:

ConstantVarDecl → TypeID{‘[‘Expression’]’} ‘=’ Expression ‘;’

For List declaration, a more compact representation is available with an array style.

`T myVar[N1][N2]...[Nn]` is equivalent to `List(type: List(type: ... List(type: T, size=Nn), ..., size=N2), size=N1) myVar` where the type is `T`.

An explicit variable declaration can take one of the following two forms, where `T` is a type, `v` an identifier that is the variable name and `E` an expression of type `T`.

— `T v:=E` — declares an assignable variable of type `T` with the value of `E` as its initial value.

— `T v=E` — declares a non-assignable variable of type `T` with the value of `E` as its value.

Variables declared in the first method are called stateful variables because they may be changed by the execution of a statement. Variables declared in the last method are referred to as stateless variables, or constants.

Explicit variable declarations may occur in the following places:

— actor state variables (with ending punctuation ";");

— the **var** block of a surrounding lexical context (with ending punctuation "," or no ending punctuation; see LocalVarDecl in D.7.3)

D.7.3 Function and procedure declaration

The general format for declaring functions and procedures is as follows:

FunDecl → ‘function’ ID’(‘ [FormalPars] ’)’ ‘-- > ‘Type
[[‘var’ VarDecls] ‘:’ Expression] ‘end’

ProcDecl → ‘procedure’ ID’(‘ [FormalPars] ‘)’
[[‘var’ VarDecls] ‘begin’ { Statement }] ‘end’

LocalVarDecls should follow a special rule: 1) if a LocalVarDecl is the last one or the only one, there shall be no any ending punctuation; 2) if a LocalVarDecl is not the last one, there shall be a comma "," as the ending punctuation (or separator between it and the next LocalVarDecl). For instance, a function declaration would look like this:

```
function timestwo (int x)--> int: 2 * x end
```

D.7.4 Name scoping

The scope of a name, whether that of a variable or that of a function or procedure, is the lexical construct that introduces it. All expressions and assignments using the name inside this construct will refer to that variable binding or the associated function or procedure, unless they occur inside some other construct that introduces the same name again, in which case the inner name shadows the outer one.

In particular, this includes the initialization expressions that are used to compute the initial values of the variables in variable declarations. Consider the following group of variable declarations inside the same construct, i.e. with the same scope:

```
n=1 + k,
k=6,
m=k * n
```

This set of declarations (of, in this case, non-assignable variables, although this does not have a bearing on the rules for initialization expression dependency) would lead to `k` being set to `6`, `n` to `7` and `m` to `42`. Initialization expressions may not depend on each other in a circular manner, e.g. the following list of variable declarations would not be well formed:

```
n=1 + k,
k=m - 36,
m=k * n
```

More precisely, a variable may not be in its own dependency set. Intuitively, this set contains all variables that need to be known in order to compute the initialization expression. These are usually the free variables of the expression itself, plus any free variables used to compute them and so on. In the last example, `k` depended on `m` because `m` is free in `m - 36`, and since `m` in turn depends on `k` and `n`, and `n` on `k`, the dependency set of `k` is `{m,k,n}` which does contain `k` itself and is therefore an error.

D.8 Expressions

D.8.1 Overview

Expressions evaluate to a value and are side-effect free, i.e. they do not change the state of the actor or assign or modify any other variable. Thus, the meaning of an expression can be described by the value it is evaluating to.

The following is an overview of the kinds of expressions and expression syntaxes provided in RVC-CAL.

Expression → Expression { ‘,’ Expression }
Expression → BinaryExpression
| SimpleExpression

SingleExpression → OperatorExpression
| ListComprehension
| ifExpression
| LetExpression
| ‘(‘ Expression ‘)’
| IndexerExpr
| ID ‘(‘ Expressions ‘)’
| ID
| ExpressionLiteral

The following subclause discusses the individual kinds of expressions in more detail.

D.8.2 Literals

Expression literals are constants of various types in the language. They look as follows:

ExpressionLiteral → IntegerLiteral | DecimalFractionLiteral
| StringLiteral
| **true** | **false**

The type of **true** and **false** is `bool`.

D.8.3 Variable references

The expression used to refer to the value bound to a variable at any given point during the execution is simply the name of the variable itself, i.e. an identifier.

D.8.4 Function application

An expression of the form $F(E_1, \ldots, E_n)$ is the application of a function to n parameters, possibly none. F is the name of a function which shall be visible at the point of this expression, and E_i are expressions of types matching the types of the parameters declared in the declaration of F.

D.8.5 Indexing

An indexing expression selects an object from a list. The general format is

IndexerExpr → ID '[' Expression ']' {'[Expressions ']'}

The expressions within the brackets are called "indices". There shall be at least one such index. If there are more than one, the list expression shall be a list of appropriate dimensionality, i.e. it shall contain other lists as elements and so forth.

D.8.6 Operators

There are two kinds of operators in RVC-CAL: unary prefix operators and binary infix operators. A binary operator is characterized by its associativity and its precedence. In RVC-CAL, all binary operators associate to the left, while their precedence is defined by the platform, and have fixed predefined values for built-in operators (which are used to work on instances of built-in types). Unary operators always take precedence over binary operators.

`a + b + c` is always `(a + b) + c`.

`#a + b` is always `(#a) + b`.

`a + b * c` is `a + (b * c)` if * has a higher precedence than +, which is usually the case.

Operators are just syntactical elements. They represent ordinary unary or binary functions, so the only special rules for operators are syntactical.

D.8.7 Conditional expressions

The simple conditional expression has the following form:

IfExpression → **if** Expression **then** Expression **else** Expression **end**

The first sub-expression shall be of type `bool`. When the first sub-expression is evaluated to **true**, the value of the second sub-expression is selected as the value of the entire expression. Otherwise, the value of the third sub-expression is selected.

The type of the conditional expression is the most specific supertype (least upper bound) of both, the second and the third sub-expression. It is undefined (i.e. an error) if this does not exist.

D.8.8 List comprehensions

List comprehensions are expressions, which construct lists. There are two variants of list comprehensions, those with and those without generators. We will first focus on comprehensions without generators, also called *enumerations*, and then turn to the more general comprehensions with generators. The reason for this order of presentation is that the meaning of comprehensions with generators will be defined by reducing them to enumerations.

D.8.8.1 Enumerations: list comprehensions without generators

The most basic form of list comprehension just enumerates the elements. Its syntax is as follows:

SimpleListComprehension → '[' [Expressions] ']'

Example: If `n` is the number 10, then the simple set expression

```
[n, n*n, n-5, n/2]
```

evaluates to the list `[10, 100, 5, 5]`.

D.8.8.2 List comprehensions with generators

Simple comprehension expressions only allow the construction of a list whose size is correlated with the size of the expression. In order to facilitate the construction of large- or variable-sized lists, RVC-CAL provides generators to be used inside an expression constructing them. The syntax looks as follows:

ListComprehension → '[' [Expressions [':' Generators] ['|' Expression]] ']'
Generators → Generator { ',' Generator }
Generator → **for** TypeID **in** Expression

The generators, which begin with the keyword, **for**, introduce new variables, and successively instantiate them with the elements of the proper list after the keyword, **in**. The expression computing that list may refer to the generator variables defined to the left of the generator it belongs to.

The optional expressions following the collection expression in a generator are called filters. They shall be of type `bool`, and only variable bindings for which these expressions evaluate to **true** are used to construct the collection.

Example:

```
[1, 2, 3]
```

is the list of the first three natural numbers. The list

```
[2 * a: for int a in[1, 2, 3]]
```

contains the values 2, 4 and 6, while the list

```
[a: for int a in [1, 2, 3] a > 1]
```

describes (somewhat redundantly) the set containing 2 and 3. Finally, the list

```
[a * b: for int a in [1, 2, 3] for int b in [4, 5, 6] b > 2 * a]
```

contains the elements 4, 5, 6, 10 and 12.

Writing the above as

```
[a * b: for int a in [1, 2, 3] b > 2 * a, for int b in [4, 5, 6]]
```

is illegal (unless `b` is a defined variable in the context of this expression, in which case, it is merely very confusing), because the filter expression `b > 2 * a` occurs before the generator that introduces `b`.

If the generator collection is a set rather than a list, the order in which elements are extracted from it will be unspecified. This may affect the result in the case of a list comprehension.

D.9 Statements

D.9.1 Overview

The execution of an action (as well as actor initialization) happens as the execution of a (possibly empty) sequence of statements. The only observable effect of a statement is a change of the variable assignments in its environment. Consequently, the meaning of a statement is defined by how the variables in its scope change due to its execution. RVC-CAL provides the following kinds of statements:

```
Statement → AssignmentStmt
            | CallStmt
            | BlockStmt
            | IfStmt
            | WhileStmt
            | ForeachStmt
```

D.9.2 Assignment

Assigning a new value to a variable is the fundamental form of changing the state of an actor. The syntax is as follows:

```
AssignmentStmt → ID [ Index ] ':=' Expression ';'
Index → '[' [ Expression ] ']' {'[' Expression ']'}
```

An assignment without an index or a field reference is a simple assignment, while one with a field reference is a field assignment, and one with an index is called an indexed assignment.

D.9.2.1 Simple assignment

In a simple assignment, the left-hand side is a variable name. A variable by that name shall be visible in this scope, and it shall be assignable.

The expression on the right-hand side shall evaluate to an object of a value compatible with the variable [i.e. its type shall be assignable to the declared type of the variable, if any (see D.6.2)]. The effect of the assignment is of course that the variable value is changed to the value of the expression. The original value is thereby overwritten.

D.9.2.2 Assignment with indices

If a variable is of a type that is indexed, and if it is mutable, assignments may also selectively assign to one of its indexed locations, rather than only to the variable itself.

In RVC-CAL, an indexed location inside an object is specified by a sequence of objects called indices, which are written after the identifier representing the variable, and which is enclosed in square brackets.

D.9.3 Procedure call

Calling a procedure is written as follows:

```
CallStmt → ProcedureSymbol '(' [ Expression ] ')'
ProcedureSymbol → ID
```

The procedure symbol shall be defined in the current context, and the number and types of the argument expressions shall match the procedure definition. The result of this statement is the execution of the procedure, with its formal parameters bound position-wise to the corresponding arguments.

D.9.4 Statement blocks (begin ... end)

Statement blocks are grouping a sequence of statements within their own nested scope, permitting the declaration of local variables valid in that scope only.

BlockStmt → **begin** [**var** LocalVarDecls **do**] { Statement } **end**

The form

```
begin var <decls> do <stmts> end
```

defines the variables in <decls> and executes the statements in <stmts> with the resulting variable bindings. The variable bindings are only visible to these statements.

D.9.5 If-statement

The if-statement is the simplest control-flow construct

IfStmt → **if** Expression **then** { Statement } [**else** { Statement }] **end**

As is to be expected, the statements following the **then** are executed only if the expression evaluates to **true**, otherwise, the statements following the **else** are executed, if present. The expression shall be of type `bool`.

D.9.6 While-statement

Iteration constructs are used to repeatedly execute a sequence of statements. A while-construct repeats execution of the statements as long as a condition specified by a `bool` expression is true.

WhileStmt → **while** Expression [**var** VarDecls] **do** [Statements] **end**

It is an error for the while-statement to not terminate.

D.9.7 Foreach-statement

The **foreach**-construct allows iterating over collections and successively binds variables to the elements of the expression with the execution of a sequence of statements for each such binding.

ForeachStmt → ForeachGenerator { ‘,’ ForeachGenerator }
[**var** VarDecls] **do** [Statements] **end**
ForeachGenerator → **foreach** TypeID **in** Expression

The basic structure and execution mechanics of the foreach-statement is not unlike that of the comprehensions with generators discussed in D.8.8.2. However, where in the case of comprehensions a collection was constructed piecewise through a number of steps specified by the generators, a foreach statement executes a sequence of statements for each complete binding of its generator variables.

Example:

The following code fragment

```
s:=0;
foreach int a in [1,2] foreach int b in[1,2] do
   s:=s + a*b;
end
```

results in `s` containing the number 9.

D.10 Actions

D.10.1 Overview

An action in RVC-CAL represents a (often large or even infinite) number of transition of the actor transition system described in D.11. An RVC-CAL actor description can contain any number of actions, including none. The definition of an action includes the following information:

— input tokens;

— output tokens;

— state change of the actor;

— additional firing conditions.

In any given state, an actor may take one of a number of transitions (or none at all), and these transitions are represented by actions in the actor description.

The syntax of an action definition is as follows:

ActionTag → [ActionTag ':'] **action** ActionHead [**do** Statements] **end**
ActionTag → ID { '.' ID }
ActionHead → InputPatterns '==>' OutputExpressions
  [**guard** Expressions] [**var** VarDecls]

Actions are optionally preceded by action tags which come in the form of qualified identifiers [i.e. sequences of identifiers separated by dots (see D.11.2)]. These tags need not be unique, i.e. the same tag may be used for more than one action. Action tags are used to refer to actions or sets of actions, in action schedules and action priority orders; see D.11 for details.

The head of an action contains a description of the kind of inputs this action applies to, as well as the output it produces. The body of the action is a sequence of statements that can change the state, or compute values for local variables that can be used inside the output expressions.

Input patterns and output expressions are associated with ports either by position or by name. These two kinds of associations cannot be mixed. So if the actor's port signature is

```
Input1, Input2 ==> ...
```

an input pattern may look like this:

```
[a], [b, c]
```

(binding `a` to the first token coming in on `Input1`, and binding `b` and `c` to the first two tokens from `Input2`). It may also look like this:

```
Input2:[c]
```

but never like this:

```
[d] Input2:[e]
```

This mechanism is the same for input patterns and output expressions.

The following subclauses elaborate on the structure of the input patterns and output expressions describing the input and output behaviour of an action, as well as the way the action is selected from the set of all actions of an actor.

In discussing the meaning of actions and their parts, it is important to keep in mind that the interpretation of actions is left to the model of computation and is not a property of the actor itself. It is

therefore best to think of an action as a declarative description of how input tokens, output tokens and state transitions are related to each other. See also D.10.5.

D.10.2 Input patterns and variable declarations

Input patterns, together with variable declarations and guards, perform two main functions: (1) they define the input tokens required by an action to fire, i.e. they give the basic conditions for the action to be fired which may depend on the value and number of input tokens and on the actor state, and (2) they declare a number of variables which can be used in the remainder of the action to refer to the input tokens themselves. The syntax is as follows:

InputPattern → [ID ':'] '[' IDs ']' [RepeatClause]
RepeatClause → **repeat** Expression

The static type of the variables declared in an input pattern depends on the token type declared on the input port, but also on whether the input pattern contains a repeat-clause.

A pattern without a repeat-expression is just a number of variable names inside square brackets. The pattern binds each of the variable names to one token, reading as many tokens as there are variable names. The number of variable names is also referred to as the *pattern length*. The static type of the variables is the same as the token type of the corresponding port.

Example (Input pattern without repeat-clause):

Assume the sequence of tokens on the input channel is the natural numbers starting at 1, i.e.:

```
1, 2, 3, 4, 5, ...
```

The input pattern `[a, b, c]` results in the following bindings:

```
a=1, b=2, c=3
```

If a pattern contains a repeat-clause, that expression shall evaluate to a non-negative integer, say N. If the pattern length is L the number of tokens read by this input pattern and bound to the L pattern variables is NL. Since in general there are more tokens to be bound than variables to bind them to (N times more, exactly), variables are bound to lists of tokens, each list being of length N. In the pattern, the list bound to the k-th variable contains the tokens numbered k, $L + k$, $2L + k$, ..., $(N - 1)L + k$. The static type of these variables is `List[T]`, where `T` is the token type of the port.

Example (Input pattern with repeat-clause):

Assume again the natural numbers as input sequence. If the input pattern is

```
[a, b, c] repeat 2
```

it will produce the following bindings:

```
a=[1,4], b=[2,5], c=[3,6]
```

D.10.3 Scoping of action variables

The scope of the variables inside the input patterns, as well as the explicitly declared variables in the var-clause of an action is the entire action. As a consequence, these variables can depend on each other. The general scoping rules from D.7 need to be adapted in order to properly handle this situation.

In particular, input pattern variables do not have any initialization expression that would make them depend explicitly on any other variable. However, their values clearly depend on the expressions in the repeat-clause (if present). For this reason, for any input pattern variable v we define the set of free variables of its initialization expression F_v to be the union of the free variables of the corresponding expressions in the repeat-clause.

The permissible dependencies then follow from the rules in D.7.

Example (Action variable scope):

The following action skeleton contains dependencies between input pattern variables and explicitly declared variables:

```
[n], [k], [a] repeat m * n ==> ...
var
   m=k * k
do ... end
```

These declarations are well formed, because the variables can be evaluated in the order `k`, `m`, `n`, `a`.

By contrast, the following action heads create circular dependencies:

```
[a] repeat a[0] + 1 ==> ... do ... end
[a] repeat n ==> ... var
   n=f(b), b=sum(a)
do ... end
```

D.10.4 Output expressions

Output expressions are conceptually the dual notion to input pattern. They are syntactically similar, but rather than containing a list of variable names which get bound to input tokens, they contain a list of expressions that computes the output tokens, the so-called token expressions.

OutputExpression → [ID ':'] '[' Expressions ']' [RepeatClause]
    RepeatClause → ***repeat*** Expression

The repeat-clause works not unlike in the case of input patterns, but with one crucial difference. For input patterns, it controls the *construction* of a data structure that was assembled from input tokens and then bound the pattern variables. In the case of output expressions, the values computed by the token expressions are themselves these data structures, and they are *disassembled* according to the repeat-clause, if it is present.

In output expressions without repeat-clause, the token expressions represent the output tokens directly, and the number of output tokens produced is equal to the number of token expressions. If an output expression does have a repeat-clause, the token expressions shall evaluate to lists of tokens, and the number of tokens produced is the product of the number of token expressions and the value of the repeat-expression. In addition, the value of the repeat-expression is the minimum number of tokens each of the lists shall contain.

Example (Output expressions):

The output expression

```
... ==>[1,2,3]
```

produces the output tokens 1, 2, 3.

The output expression

```
... ==> [[1,2,3],[4,5]] repeat 2
```

produces the output tokens 1, 2, 4, 5.

D.10.5 On action selection: guards and other activation conditions

At any given point during the execution of an actor, an action may potentially fire on some input data. Whether it is activated, i.e. whether in a given situation it actually can fire, depends on whether its activation conditions have all been met. The minimal conditions are as follows:

a) according to the action schedule (see **Legal action sequence** in D.11.3), this action may fire next;

b) no higher-priority action is activated (see D.11.4);

c) there are sufficient input tokens available to bind the input pattern variables to appropriate values;

d) given such a binding, all guard expressions (which shall be Boolean expressions) evaluate to true.

D.10.6 Initialization actions

Initialization actions are executed at the beginning of an actor's life cycle. They are very similar to regular actions, with two important differences.

a) Since the assumption is that at the beginning of an actor execution no input is available, initialization actions have no input patterns; however, they may produce output.

b) With the exception of initialization expressions in variable declarations, an initialization action contains the first code to be executed inside the actor. Any state invariants in the actor may not hold, and instead have to be established by the initialization action.

The syntax of initialization actions is as follows:

InitializationAction → [ActionTag ':']
  initialize InitializerHead [**do** Statements] **end**
InitializerHead → '==>' OuputExpressions
  [**guard** Expressions] [**var** VarDecls]

The activation conditions for actions apply also to initialization action, of course, since there is no input, the conditions concerning input tokens become vacuously true.

If an actor should have more than one initialization action, and if more than one is activated at the beginning of an actor execution, one of them is chosen arbitrarily.

D.11 Action-level control structures

D.11.1 Overview

In RVC-CAL, an action expresses a relation between the state of an actor and input tokens, and the successor state of the actor and output tokens. In general, RVC-CAL actors may contain any number of actions, and in a given situation, any subset of those may be ready to be executed. For example, both actions of the following actor may be able to execute, if there is a token available on either input port:

Example (Nondeterministic Merge):

```
actor NDMerge () A, B ==> C:
   action A:[x] ==> [x] end
   action B:[x] ==> [x] end
end
```

It is important to emphasize that the policy used to choose between the two actions above is not part of the actor specification. This flexibility may be desired, but sometimes the actor writer may want to have more control over the choice of the action, e.g. if the Merge actor is supposed to alternate between reading its input ports, one might use actor state to realize this behaviour:

Example (Basic FairMerge):

```
actor FairMerge () A, B ==> C:
   s:=1;

   action A:[x] ==> [x]
     guard s=1
     do
          s:=2;
     end

     action B:[x] ==> [x]
     guard s=2
     do
          s:=1;
     end
end
```

This way of specifying action choice has two key drawbacks. First, it is very cumbersome to write and maintain, and it does not scale very well even for modest numbers of actions and states. Furthermore, this way of specifying action choice essentially obfuscates the "real" logic behind guards, state variable and assignments, so that it becomes harder to extract the intent from the actor description, both for tools and for human readers.

These are the key motivations for using action schedules, i.e. structured descriptions of possible orders in which actions may fire. Before we can discuss action schedules in D.11.3, we need to take a closer look at how actions are referred to inside of them.

D.11.2 Action tags

Actions are optionally prefixed with action tags (see D.10.1), which are qualified identifiers:

ActionTag → QualID

QualID → ID{ '.' ID}

The same tag may be used for more than one action. In the following, we write the set of all actions tagged by a tag t as $\bar{t}$, and the tag of some action a as t_a. The empty tag is written as ε, and the set of all untagged actions is therefore $\bar{\varepsilon}$.

Action tags are ordered by a prefix ordering: We say that $t \subseteq t'$, i.e. t is a prefix of t', if t' starts with all the identifiers in t in the same order, followed by any number of additional identifiers, including none. For instance, $a.b.c \subseteq a.b.c.x$ and $a.b \subseteq a.b$, but $a.b \not\subset a.c$. We call t' an extension of t.

When used inside action schedules and priority orderings, a tag denotes the set of actions, which are labelled with tags that are extensions of it. For any tag t this set is called $\hat{t}$ and is defined as follows:

$$\hat{t} =_{\text{def}} \{a | t \subseteq t_a\}$$

D.11.3 Action schedules

Action schedules are structured descriptions of possible sequences in which the actions of an actor may be executed. A finite state machine specifies these sequences. In general, the set of possible sequences may be finite or infinite and any specific sequence may also be finite or infinite.

An action schedule effectively describes a (regular) language L in the alphabet of action tags. This language is used to constrain the legal sequences of action firings as follows.

Legal action sequence. Given a tag language L, assume a finite sequence of actions $(a_i)_{i=1...n}$ and a sequence $(b_j)_{j=1...m}$ with $m \le n$ and a strict monotonic function $f: \{1...m\} \to \{1...n\}$ such that the following holds for all $j \in \{1..m\}$ and $i \in \{1..n\}$:

— $b_j = a_{f(j)}$;

— $t_{b_j} \neq \varepsilon$;

— $t_{a_i} \neq \varepsilon, \forall i \notin f^{-1}\left[\{1..m\}\right]$.

In other words, (b_j) is the subsequence in (a_i) with non-empty tags. If (b_j) is empty, then (a_i) is a legal action sequence.

If (b_j) is not empty, then (a_i) is a legal action sequence, if and only if, there exists a sequence of tags $(t_j)_{j=1...m}$, such that the following holds:

— for all $j \in \{1...m\}, b_j \in \hat{t}_j$;

— there exists a $w \in L$ such that $(t_j) \subseteq w$.

A consequence of this definition is that untagged actions may occur at any point in the schedule — conversely, schedules do not constrain untagged actions in any way.

The following paragraph describes a tag language based on finite state machines.

A finite state machine schedule defines a number of transitions between states (and an initial state) that are each labelled with one or more action tags.

ScheduleFSM → **schedule** [**fsm**] ID ';'
  { StateTransition ';' }
  end

StateTransition → ID '(' ActionTag ')' '-->' ID

The state before the colon is the initial state, and all states are accepting. The tag language is the set of all sequences of tags that label transitions leading from the initial state to any other state of the finite state machine.

Several transitions starting from the same state may be written as separated by the '|' character. The following illustrates the use of a finite state machine action schedule to express the `FairMerge` actor somewhat more concisely.

Example (FairMerge, with FSM schedule):

```
actor FairMerge1 () A, B ==> C:

   InA: action A:[x] ==> [x] end
     InB: action B:[x] ==> [x] end

     schedule fsm WaitA:
          WaitA (InA) --> WaitB;
          WaitB (InB) --> WaitA;
     end
end
```

D.11.4 Priorities

Priorities are very different from action schedules, in that, they genuinely add to the expressiveness of RVC-CAL. It would not be possible, in general, to reduce them to existing constructs in the way schedules can in principle be reduced to a state variable and guards/assignments. Among other things,

priorities allow actors to effectively test for the absence of tokens. As a consequence, actors can express non-prefix monotonic processes, which are powerful, but at the same time can be dangerous, because it means the results computed by an actor may depend on the way it was scheduled with respect to the other actors in the system.

Priorities are defined as a partial order relation over action tags, which induce a partial order relation over the actions. An action can only fire if there is no other enabled action that is higher in this partial order. The order is specified as follows:

PriorityOrder → **priority**{ PriorityInequality ';'} **end**

PriorityInequality → ActionTag '>' ActionTag { '>' ActionTag }

The priority inequalities are specified over tags, i.e. they have the form $t_1 > t_2$. These inequalities induce a binary relation on the actions are follows:

$$a_1 > a_2 \Leftrightarrow \exists t_1, t_2 : t_1 > t_2 \wedge a_1 \in \hat{t}_1 \wedge a_2 \in \hat{t}_2$$

$$\vee \exists a_3 : a_1 > a_3 \wedge a_3 > a_2$$

The priority inequalities are valid if the induced relation on the actions is in non-reflexive partial order, i.e. it is asymmetric and transitive. Transitivity follows from the definition, but asymmetry and non-reflexivity do not. In fact they do not even follow if the relation on the tags is a partial order. Consider the following example:

```
A.B > X > A
```

This is obviously a proper order on the tags. However, if we have two actions labelled `X` and `A.B`, then the induced relation is clearly not asymmetric, hence the system of priority inequalities is invalid.

With priorities, we can express a Merge actor that prefers one input over the other like this:

Example (BiasedMerge):

```
actor BiasedMerge () A, B ==> C:
   InA: action A:[x] ==> [x] end
   InB: action B:[x] ==> [x] end

   priority
      InA > InB;
   end
end
```

Perhaps more interestingly, we can express a merge actor that is fair, in the sense that it will consume equal amounts of tokens from both inputs as long as they are available, but will not halt due to lack of tokens on only one of its input ports. It is also non-deterministic, i.e. it does not specify the order in which it outputs the tokens.

Example (FairMerge, with priorities):

```
actor FairMerge3 () A, B ==> C:
     Both: action[x], [y] ==> [x, y] end
   Both: action[x], [y] ==> [y, x] end
   One: action A:[x] ==> [x] end
   One: action B:[x] ==> [x] end

   priority
        Both> One;
   end
```

D.12 Basic runtime infrastructure

D.12.1 Overview

This subclause describes the basic runtime infrastructure, i.e. the kinds of objects and operations on them, that implementations shall provide in order to implement RVC-CAL. A list of keywords is also included in this subclause.

D.12.2 Operator symbols

D.12.2.1 Unary operator symbols

The following table summarizes the predefined unary operator symbols in RVC-CAL.

Operator	Operand type(s)	Meaning
not	bool	logical negation
#	List(...)	number of elements
–	number	arithmetic negation

D.12.2.2 Binary operator symbols

The following table lists the predefined binary operator symbols in the RVC-CAL language. They are sorted by increasing binding strength. Their binding strength is given by a precedence figure P, higher precedence binds stronger.

P	Operator	Operand 1	Operand 2	Meaning
1	and	bool	bool	logical conjunction
	or	bool	bool	logical disjunction
2	=	any	any	equality
	!=	any	any	inequality
	<	number	number	less than
	<=	analogous to <		less than or equal
	>	analogous to <		greater than
	>=	analogous to <		greater than or equal
3	+	number	number	addition
		List[T]	List[T]	concatenation
	–	number	number	difference

P	Operator	Operand 1	Operand 2	Meaning
4	div	number	number	integral division
	mod	number	number	modulo
	*	number	number	multiplication
	/	number	number	division
5	..	int	int	integral list from Operand 1 to Operand 2 (inclusive)
6	&	int	int	bitwise and
	|	int	int	bitwise or
	^	int	int	bitwise xor
	~	int	int	bitwise not
	<<	int	int	left bit shift
	>>	int	int	right bit shift

D.12.3 Keywords

action	actor	and	begin	div	do	else	end	false	for
foreach	fsm	function	or	guard	if	in	initialize	mod	not
priority	procedure	repeat	schedule	then	true	var	while		

Annex E
(informative)

FU Classification according to their dataflow model of computation of RVC-CAL

E.1 Overview

This annex describes conditions used to classify FUs, so that programmers and RVC codec implementers can make sure that an FU is classified correctly by analysis and translation tools.

E.2 Basic terms

E.2.1 Token rates

An FU consists of any number of actions of the form

```
action A:[a1, a2], B:[b]
       ==> X:[s + a1, s + a2, s], Y:[a1 + b]

guard P(b) do

   s:=s + b;

end
```

The constructs preceding the ==> are called *input patterns*, and establish how many tokens the action requires in order to be enabled on each port, which is also the number of tokens it will consume when it fires. For instance, the construct A: [a1, a2] specifies that the action requires two tokens on port A in order to fire. The constructs following the ==> are called *output expressions*, and they specify how many tokens are being produced on each output port when the action fires (as well as their values, of course). For instance, the expression X: [s + a1, s + a2, s] implies that three tokens will be produced on output X, since that is the number of comma-separated expressions between the square brackets.

Together, input patterns and output expressions define the *token rates* of an action, which is the number of tokens produced and consumed whenever that action fires. In the example above, the action will consume two tokens on input port A, one on input port B, and it will produce three tokens on output port X and one on output port Y. If the FU has no any other ports besides these four, the rate of token consumption or production on those ports for this action is zero.

E.2.2 Priorities

Actions within an FU may be related to each other through a priority statement such as the following:

```
priority
    A1 > A2;
    A2 > A3;
    A2 > A4;
end
```

The names A1, A2, A3 and A4 are *action tags*, identifying one or more actions. The order among actions is *partial*. Note that in the example, A3 and A4 are of lower priority then A2 (and, by implication, A1), but

they are related to each other. Also, if the FU in the example above contains another action tagged, e.g. A5, that action will not be related to any of the actions tagged A1, A2, A3 and A4.

A special case of priority orders is the *total order*, in which for any pair of two actions one of them has a higher priority than the other. For instance, if the example above is modified as follows, the tags are totally ordered:

```
priority
    A1 > A2;
    A2 > A3;
    A3 > A4;
end
```

For all actions to be totally ordered within an FU, each tag uniquely identifies one action, i.e. no two actions may be tagged by the same name.

E.2.3 Cyclic scheduler

A scheduling finite state machine is considered to be a *cyclic scheduler* if there is a sequence of states s_1, ..., s_n such that

— s_1 is the initial state of the scheduler, and

— there are exactly n transitions, viz. s_i to s_{i+1} for all i from 1 to $n-1$, and s_n to s_1.

The length of the cycle is n. If there is no scheduler, the absent scheduler is considered to be (trivially) cyclic with cycle length 1. For instance, the following scheduler is cyclic, with cycle length 2:

```
    schedule fsm s0:
        s0 (A) --> s1;
        s1 (B) --> s0;
    end
```

By contrast, the next scheduler is not cyclic:

```
    schedule fsm s0:
        s0 (A) --> s1;
        s0 (B) --> s2;
        s1 (C) --> s0;
        s2 (C) --> s0;
    end
```

E.3 FU classes and taxonomy

In the following, four specialized classes of FUs can be distinguished in order of specialization: non-deterministic FUs [following Dataflow Process Network semantic (DPN)], prefix-monotonic FUs [also known as Kahn Process Network semantic (KPN)], Cyclo-static dataflow (CSDF) FUs and synchronous dataflow (SDF) FUs. Each class properly includes the subsequent classes, i.e. all prefix-monotonic FUs are also deterministic.

The rules provided to characterize the FU classes are conservative. If an FU conforms to the rule, it is guaranteed to belong to the corresponding class, but there may be FUs that do *not* conform to the rule and still be members of the class.

E.3.1 Non-deterministic FUs

E.3.1.1 Non-determinism caused by time dependency

Rule: Time dependency occurs when the following conditions are met.

a) When two actions are related to each other by priority order (i.e. one has a higher priority than the other) and the input pattern of the lower priority action is a strict subset of the input pattern of the higher priority action. The input pattern is defined in D.9.2.

b) The above two actions have guards that are not mutually exclusive.

Example:

```
actor BiasedMergeTimeDependent () int A, int B ==> int X:

    CopyA: action A:[v] ==> X:[v] end
    CopyB: action B:[v] ==> X:[v] end

    priority CopyA > CopyB; end
end
```

The following *ClipTimeDependent* FU is described with a time-dependent behaviour.

Example:

```
actor ClipTimeDependent () int I, bool SIGNED ==> int O:
  int count:=-1;

  read_signed: action SIGNED:[s] ==>
  guard count < 0
  do
    count:=63;
  end

  limit: action I:[i] ==> O:[f_clip(X)]
  do
    count:=count - 1;
  end
```

The *ClipTimeDependent* FU can have no time-dependent behaviour as described in the following *Clip* FU.

Example:

```
actor Clip() int I, bool SIGNED ==> int O:
  int count:=-1;

  read_signed: action SIGNED:[s] ==>
  guard count < 0
  do
    count:=63;
  end

  limit: action I:[i] ==> O:[f_clip(X)]
  guard count >= 0
  do
    count:=count - 1;
  end
```


E.3.1.2 Non-determinism caused by absence of priorities

Rule: An FU becomes *non-deterministic* when there exist actions that are eligible for firing at the same time and where those actions are not totally ordered.

Example:

```
actor Split () int A ==> int X, int Y:

  action A:[v] ==> X:[v] end
  action A:[v] ==> Y:[v] end
end
```

E.3.2 Deterministic FUs

E.3.2.1 Kahn process FUs — Prefix-monotonicity

Rule: An FU is *prefix-monotonic* (also known as a *Kahn process*) if it is deterministic and is not time-dependent.

Example:

```
actor PingPongMerge () int A, int B ==> int X:

    CopyA: action A:[v] ==> X:[v] end
    CopyB: action B:[v] ==> X:[v] end
    schedule fsm s0:
        s0 (CopyA) --> s1;
        s1 (CopyB) --> s0;
    end
end
```

This FU is deterministic because the scheduler guarantees that only one action is eligible in each state.

Example:

```
actor Foo () int A ==> int X:

    Hi: action A:[v] ==> X:[v]
        guard v > 0 end
    Lo: action A:[v, w] ==> X:[v + w] end

    priority Hi > Lo; end
end
```

Here, the higher priority action reads one token from the input port, while the lower priority action requires two.

E.3.2.2 Cyclo-static dataflow (CSDF) FUs

Rule: An FU is a *cyclo-static dataflow (CSDF)* FU if it is prefix-monotonic, has a cyclic scheduler, and for each scheduler state, all eligible actions have the same token rates.

Example:

```
actor PingPongMerge () int A, int B ==> int X:

    CopyA: action A:[v] ==> X:[v] end
    CopyB: action B:[v] ==> X:[v] end

    schedule fsm s0:
        s0 (CopyA) --> s1;
        s1 (CopyB) --> s0;
    end
end
```

In this FU, only one of the two actions is eligible in each state, so the condition of all eligible actions having the same token rates is trivially fulfilled.

Example:

```
actor Bar () int A, int B ==> int X:

    A1: action A:[v] ==> X:[v]
        guard v >= 0 end
    A2: action A:[v] ==> X:[-v] end
    A3: action B:[v] ==> X:[v] end

    schedule fsm s0:
        s0 (A1, A2) --> s1;
        s1 (A3) --> s0;
    end

    priority A1 > A2; end
end
```

Here, two actions are eligible in state s0, but they are totally ordered (determinism) and have identical token rates, which makes the FUs prefix-monotonic and at the same time a cyclo-static dataflow FUs.

Counterexample:

```
actor Foo () int A ==> int X:

    Hi: action A:[v] ==> X:[v]
        guard v > 0 end
    Lo: action A:[v, w] ==> X:[v + w] end

    priority Hi > Lo; end
end
```

In this FU, the two actions, which are always eligible, have different token rates (they consume a different number of tokens on port A); therefore, this FUs is not a cyclo-static dataflow FU.

E.3.2.3 Synchronous dataflow (SDF) FUs

Rule: An FU is a *Synchronous Dataflow (SDF)* FU if it is a cyclo-static dataflow FU and all actions have the same token rates.

In principle, it should not be necessary to require an SDF actor to be cyclo-static, but in order to maintain consistency with the traditional classification, CSDF is considered as a specialized form of SDF.

Since it does not make much sense to explicitly write down cyclic schedulers with a cycle length of 1, SDF FUs usually do not contain any schedulers. Indeed, all SDF FUs can be written as an FU with a single action, and all single-action FUs are always SDF FUs if the action does now contain a guard.

Example:

```
actor E () int A, int B ==> int C, int D:
    action A:[a1, a2] B:[b] ==>
           C:[a1 + a2, a1 - a2], D:[a1 + b, a2 + b] end
end
```

This FU consumes two tokens on A, one on B, and produces two tokens on both C and D each time it fires.

Example:

```
actor PingPongPick () A, B ==> X:

    CopyA: action A:[v], B:[w] ==> X:[v] end
    CopyB: action A:[v], B:[w] ==> X:[w] end

    schedule fsm s0:
        s0 (CopyA) --> s1;
        s1 (CopyB) --> s0;
    end
end
```

This example shows that SDF FUs may have schedules with cycles longer than 1.

Counterexample:

```
actor PingPongMerge () A, B ==> X:

    CopyA: action A:[v] ==> X:[v] end
    CopyB: action B:[v] ==> X:[v] end

    schedule fsm s0:
        s0 (CopyA) --> s1;
        s1 (CopyB) --> s0;
    end
end
```

The two actions in this CSDF FU have different token rates; therefore, it is not an SDF FU.

