INTERNATIONAL STANDARD

ISO/IEC 23001-4

Second edition
2011-12-15

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 23001-4:2011(E)

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

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.

ISO/IEC 23001-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This second edition cancels and replaces the first edition (ISO/IEC 23001-4:2009) which has been technically revised.

ISO/IEC 23001 consists of the following parts, under the general title *Information technology — MPEG systems technologies*:

— *Part 1: Binary MPEG format for XML*

— *Part 2: Fragment request units*

— *Part 3: XML IPMP messages*

— *Part 4: Codec configuration representation*

— *Part 5: Bitstream Syntax Description Language (BSDL)*

— *Part 7: Common encryption in ISO base media file format files*

Introduction

This part of ISO/IEC 23001 defines the methods capable of describing codec configurations in the so-called 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 video tool library. Part 4 of ISO/IEC 23002 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.

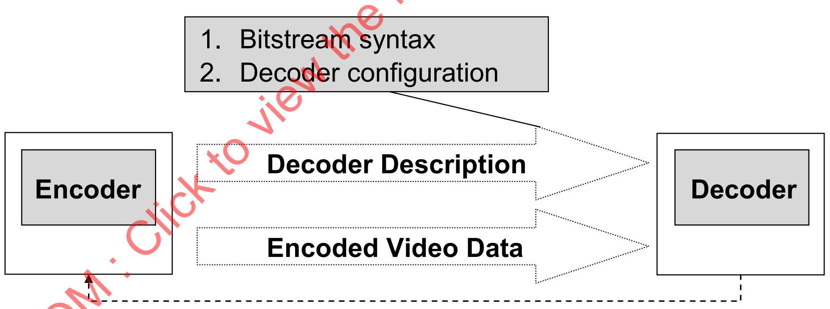


Figure 1 — Conceptual diagram of RVC

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

A more detailed description of the RVC decoder is shown in Figure 2. As shown in Figure 2, the decoder description is required for the configuration of a 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 behavioral 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 E.

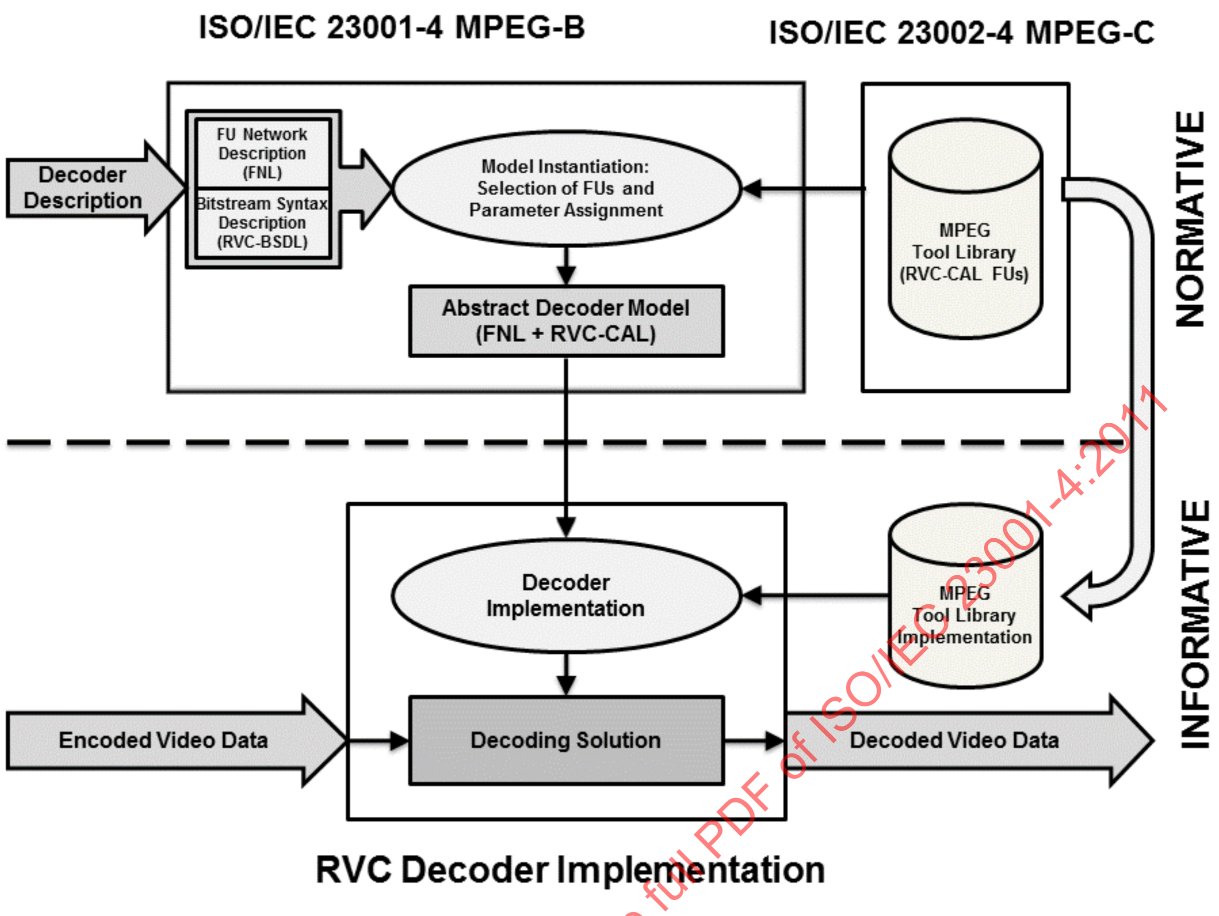

ISO/IEC 23001-4 MPEG-B
ISO/IEC 23002-4 MPEG-C
Decoder Description
FU Network Description (FNL)
Bitstream Syntax Description (RVC-BSDL)
Model Instantiation: Selection of FUs and Parameter Assignment
Abstract Decoder Model (FNL + RVC-CAL)
MPEG Tool Library (RVC-CAL FUs)
NORMATIVE
Decoder Implementation
MPEG Tool Library Implementation
INFORMATIVE
Encoded Video Data
Decoding Solution
Decoded Video Data
RVC Decoder Implementation

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 part of ISO/IEC 23001;

— 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 part of ISO/IEC 23001.

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' behavior. The normative specification of RVC-CAL is provided in this part of ISO/IEC 23001. The ADM is the behavioral 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 behavioral model. In other words what is normative is the input/output behavior of the complete ADM as well as the input/output behavior of all the FUs that are included in the ADM.

Information technology — MPEG systems technologies —

Part 4:
Codec configuration representation

1 Scope

This part of ISO/IEC 23001 defines the methods and general principles capable of describing codec configurations in the so-called 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 referenced documents are indispensable for the application 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 23001-5:2008, *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*

3 Terms and definitions

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

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

3.2
BSD
bitstream syntax description
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**] to define the parser **functional unit** (3.8)

NOTE The BSD is expressed using **reconfigurable video coding-bitstream syntax description language** (3.14).

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

NOTE 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 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 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 behavioral model [i.e. **abstract decoder model** (3.1)] an RVC **decoder description** (3.6) needs to make reference to the behavior of each FU that is provided in terms of I/O behavior by the **MPEG video tool library** (3.12) specified in ISO/IEC 23002-4.

3.6
DD
decoder description
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
FU
functional unit
modular tool which consists of a processing unit characterized by the input/output behavior

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

3.10
FNL
FU network language
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 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
RVC
reconfigurable video coding
framework defined by MPEG to promote coding standards at tool-level while maintaining interoperability between solutions from different implementers

3.14
RVC-BSDL
reconfigurable video coding-bitstream syntax description language
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
VTL
video tool library
collection of **functional units** (3.8)

4 Functional unit network description

4.1 Introduction

The FUs in MPEG RVC are specified by:

- The textual description in ISO/IEC 23002-4.
- The RVC-CAL reference software. The RVC-CAL language is formally specified in Annex D.

The Functional Unit Network Language (FNL) is formally specified in this Clause 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).

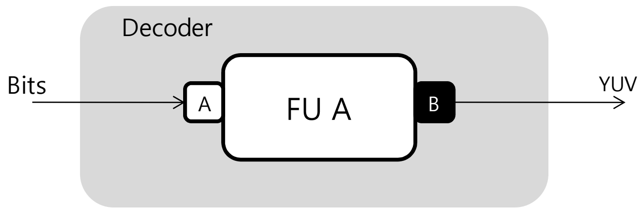


Figure 3 — FU network of one FU

A network of FUs is described in FND. An FND includes the list of the selected FUs to form the decoder and the three types of connections that are 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 the selected FUs (Figure 4) is described in FND according to the following table. 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 the following table.

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" />`

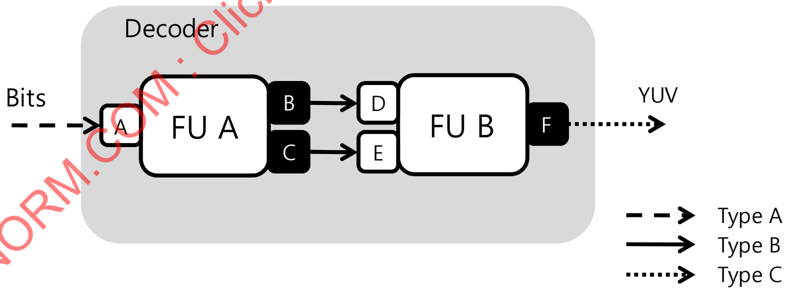


Figure 4 — Three types of connections in an FU network

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>
```

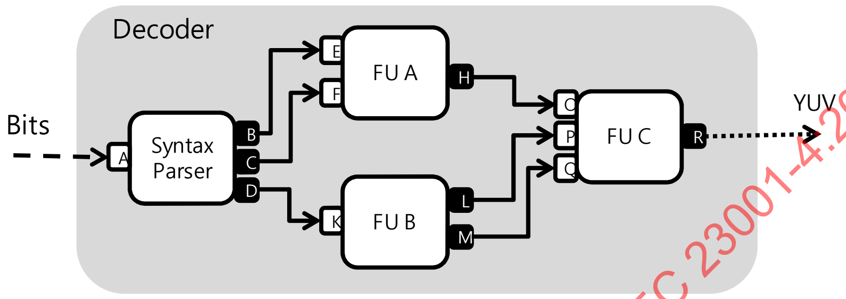


Figure 5 — Another example of FU networks

4.2 The 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 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 of this document. 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 E.

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:

- The video tool library
- The FU network description
- The 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.

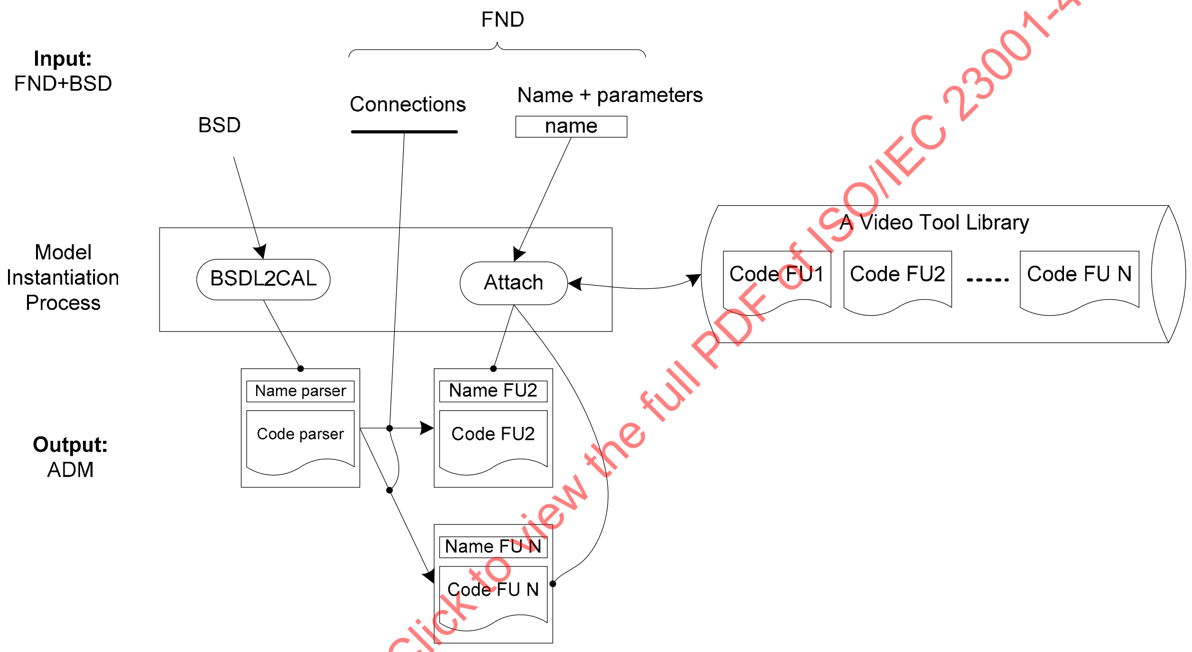


Figure 6 — Description of the model instantiation process

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="…">`. The tag `<class name="…">` indicates the name of the FU (in the video tool library) from which the FU of the ADM must be instantiated. The tag `<parameter>` provides the values of the parameters, which must 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 E 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 must 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.

Annex A
(normative)

Functional unit network description

A.1 Elements of a functional unit network

XDF — An FU network is identified by the root element called XDF that 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>
```

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). That 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>
```

Decl[@kind="Param"] — 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"/>
```

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>
```

Port — 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"/>
```

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>
```

Connection — Represents a directed connection between two ports within the network. The source of that 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 “”, 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 “”, 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

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.

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.

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"/>
```

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"/>
```

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"/>
```

Expr[@kind="Literal"][@literal-kind="String"] — String literals.

- required attribute: @value, the string value.

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

Expr[@kind="Literal"][@literal-kind="Character"] — Character literals.

- required attribute: @value, the character value.

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

Expr[@kind="List"] — This expression is a list.

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

Expr[@kind="Var"] — This expression is a variable reference.

- required attributes: @name, the name of the variable referred to.

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

Expr[@kind="Application"] — This kind of 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>
```

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>
```

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 — 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

Args — This element contains the arguments of a function application.

- required children: Expr, the argument expressions.

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

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).

Type — 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>
```

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>
```

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

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>
```

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
```

FUID — A simple identifier.

- required attribute: @id, the identifier.

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

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"/>
```

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 Introduction

This Annex provides some examples of how a 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 RVC 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 5 FUs taken from the MPEG RVC toolbox, the connections between FU and between the network and the outside world.

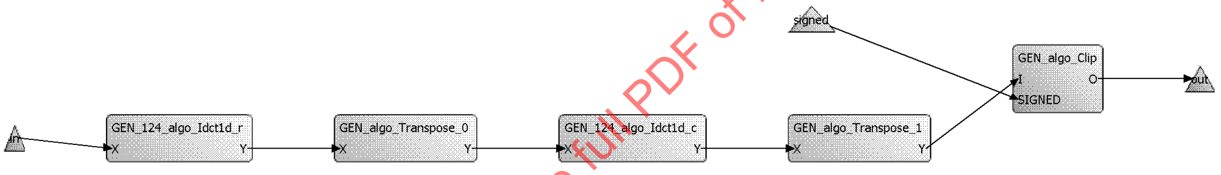


Figure B.1 — Example of networks of FU expressed using RVC FNL

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

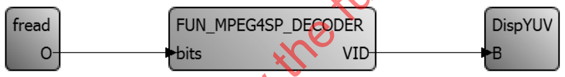
```
<?xml version="1.0" encoding="UTF-8"?><XDF name="idct2d">

   <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



```
 <?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 Introduction

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 23001-5:2008) 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 Clauses describe the specificity of the subset and the extensions of BSDL standard as specified in ISO/IEC 23001-5:2008, which are needed to obtain the RVC-BSDL used in this Part of ISO/IEC 23001 (i.e. the RVC framework).

C.2 Use of prefixes in RVC-BSDL schema

Prefixes and the corresponding namespaces are specified in RVC BSDL schema.

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

Prefix	Corresponding Namespace
xsd	http://www.w3.org/2001/XMLSchema
bs0	urn:mpeg:mpeg21:2003:01-DIA-BSDL0-NS
bs1	urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS
bs2	urn:mpeg:mpeg21:2003:01-DIA-BSDL2-NS
rvc	urn:mpeg:2006:01-RVC-NS

C.3 Constructs of RVC-BSDL

C.3.1 Introduction

This Subclause describes which 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 23001-5:2008.

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 or not 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
xsd:string	No
xsd:normalizedString	No
xsd:float	No
xsd:double	No
xsd:base64Binary	No

Table C.3 — List of BSDL built-in data types supported or not 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
bs1:bitstreamSegment	No
bs1:stringUTF16NT	No
bs1:stringUTF8	No
bs1:stringUTF16BENT	No
bs1:stringUTF16LENT	No
bs1:stringUTF8NT	No
bs1:stringUTF16	No
bs1:stringUTF16BE	No
bs1:stringUTF16LE	No
bs1:longLE	No
bs1:intLE	No
bs1:shortLE	No
bs1:unsignedLongLE	No
bs1:unsignedIntLE	No
bs1:unsignedShortLE	No
bs1:unsignedExpGolomb	No
bs1:signedExpGolomb	No

C.3.2.2 Additional data types

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. Specific functional units available in the RVC toolbox can be used to decode such portions of the bitstream before continuing the parsing. The data type `rvc:ext` indicates a portion of bitstream that needs to be decoded by an external Functional Unit. A communication scheme (described in E.1.2) is set up to make the link with this external Functional Unit. The `rvc:port` attribute helps in making this link by specifying the name of the ports used to connect the parser and the Functional Unit. An example of Variable Length Decoding is provided below:

```
<xsd:element name="DCTCoefficient" type="rvc:ext" rvc:port= "MPEG4_part2_B16"/>
```

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 shown in details in E.1.2 (Implementing Variable-Length Decoding). The `rvc:ext` type can be only applied to an `xsd:element` element.

C.3.3 Supported elements

This Subclause describes which BSDL facets are supported in RVC-BSDL within the RVC framework. The allowed BSDL-2 elements are described in Table C.4. The allowed BSDL-1 elements are described in Table C.5. The allowed XML built-in elements are reported in Table C.6.

Table C.4 — The BSDL-2 elements supported or not supported by RVC-BSDL

Element name	Supported by RVC-BSDL?
bs2:length	Yes (see C.4.3.11)
bs2:bitLength	Yes (see C.4.3.10)
bs2:startCode	Yes (see C.4.3.12)
bs2:endCode	No
bs2:escape	No
bs2:cdata	No
bs2:log2()	No
bs2:ifUnion	Yes (see C.4.3.14)
bs2:parameter	No
bs2:xpathScript	No
bs2:variable	Yes (see C.4.3.5)

Table C.5 — The BSDL-1 elements supported or not supported by RVC-BSDL

Element name	Supported by RVC-BSDL?
bs1:script	Yes

Table C.6 — The XML standard elements supported or not supported by RVC-BSDL

Element name	Supported by RVC-BSDL?
xsd:sequence	Yes (see C.4.3.3)
xsd:choice	Yes (see C.4.3.4)
xsd:all	No
xsd:group	Yes (see C.4.3.2)
xsd:element	Yes (see C.4.3.1)
xsd:simpleType	Yes (see C.4.3.6)
xsd:complexType	No
xsd:maxExclusive	No
xsd:fixed	No
xsd:annotation	Yes (see C.4.3.7)
xsd:appinfo	Yes (see C.4.3.8)
xsd:MinOccurs	No
xsd:MaxOccurs	No
xsd:default	No
xsd:union	Yes (see C.4.3.13)
xsd:length	Yes (see C.4.3.11)

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. The allowed BSDL-1 attributes are described in Table C.7. 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.7 — The BSDL-1 attributes supported or not supported by RVC-BSDL

Attribute name	Supported by RVC-BSDL?
bs1:bitstreamURI	No
bs1:ignore	No
bs1:addressUnit	No
bs1:codec	No
bs1:requiredExtensions	No
bs1:insertEmPrevByte	No
bs1:bsdlVersion	No

Table C.8 — The BSDL-2 attributes supported or not supported by RVC-BSDL

Attribute name	Supported by RVC-BSDL?
bs2:nOccurs	Yes
bs2:if	Yes
bs2:ifNext	Yes
bs2:rootElement	Yes
bs2:ifNextMask	No
bs2:ifNextSkip	No
bs2:removeEmPrevByte	No
bs2:layerLength	No
bs2:assignPre	No
bs2:assignPost	No
bs2:bsdlVersion	No
bs2:requiredExtensions	No
bs2:startContext	No
bs2:stopContext	No
bs2:redefineMarker	No
bs2:position	Yes

Table C.9 — The XML attributes supported or not supported by RVC-BSDL

Attribute name	Supported by the RVC framework?
minOccurs	No
maxOccurs	No
fixed	Yes

C.3.4.2 Additional attribute

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. Such attribute is:

`rvc:port`.

The `rvc:port` attribute is used to indicate that the corresponding element of syntax must be available outside the parser for further processing operated by the network of FUs. This attribute is applied to `xsd:element` only. An example is given below:

```
<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 must contain the description of a link connecting the output port "width" of the parser and an input port of an FU.

C.4 Syntax of RVC-BSDL

C.4.1 Introduction

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 $A \mid B$.

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: $\text{As} \rightarrow \text{A} \{ \text{','}\ \text{A} \}$

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 for elements and attributes

This Subclause describes the syntax of the element and its associated attributes.

C.4.3.1 xsd:element

This element is used to define an element of syntax.

```
<xsd:element
   name = "string"
   type = "(bs1:b1 - bs1:b32 | rvc:ext | bs1:align8 | bs1:align16 | bs1:align32
| user-defined type)"
   bs0:variable = "( true | false)"
   bs2:if = "Expression"
   bs2:ifNext = "Expression"  bs2:nOccurs = "Expression"fixed =
"hexadecimalValue"
   rvc:port = "portName"
>
Children: xsd:annotation
</xsd:element>
```

Remarks:

- The attribute `rvc:port` specifies the name of the output port to which the FU is connected. For more information about the communication between the parser and the FU, see E.1.2.
- The `rvc:port` attribute is compulsory when dealing with an element of the type `rvc:ext`.
- The user-defined type must be defined using a `xsd:simpleType` element.

C.4.3.2 xsd:group

The xsd:group element is used to define a set of elements of syntax. This element allows having a hierarchical bitstream description. There are two ways of using the xsd:group element: the definition of the group and its call.

- Definition of the group:

```
<xsd:group
   name = "string"
>
Children: xsd:sequence
</xsd:group>
```

- Call of the group:

```
<xsd:group
   ref = "string"
   bs2:if = "Expression"
   bs2:ifNext = "Expression"
   bs2:nOccurs = "Expression"
>
Children: none
</xsd:group>
```

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. 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.

C.4.3.3 xsd:sequence

According to the parent element in which this element is called, there are several possibilities:

- If the top parent element is a `xsd:group` element (in the case of the definition of the `xsd:group` element):

```
<xsd:sequence>
  Children: {sequence, group, element}
</xsd:sequence>
```

- If the top parent element is an `xsd:sequence` element, the `xsd:sequence` element is used to gather a list of consecutive elements of syntax which have conditions in common.

```
<xsd:sequence
  bs2:if = "Expression"
  bs2:ifNext = "Expression"
  bs2:nOccurs = "Expression"
>
Children: {sequence, group, element}
</xsd:sequence>
```

Remark: one can use a single or several attributes on the same `xsd:sequence` element but defining no attribute is meaningless.

Example: the elements `requested_upstream_message_type` and `newpred_segment_type` exists only if the variable `newpred_enable` equals to “1”.

```
<xsd:sequence bs2:if="$myns: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.4.3.4 xsd:choice

This element is used to make a choice between two or several elements of syntax.

```
<xsd:choice>
Children: {xsd:group, xsd:element}
</xsd:choice>
```

Remark: the `xsd:group` and `xsd:element` elements must have a `bs2:if` or `bs2:ifNext` attribute in order to be able to decide which element must be chosen. The condition on each element must be defined such as only one choice must be possible, like in the example below:

Example:

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

C.4.3.5 bs2:variable

It is common to keep in a temporary memory some elements of the bitstream syntax, according to which other elements of syntax are decoded. It is also possible to define “local” variables to keep track of some information while parsing the bitstream, This can be done using the `bs2:variable` element and the `bs0:variable` attribute.

```
<bs2:variable
  name = "string"
  value = "Expression"
>
Children: none
</bs2:variable>
```

Examples: `The bs2:variable` element defines a new variable, independent from the element of syntax being decoded. The `bs2:variable` element applies only on an `xsd:element` element. When a variable is set using this attribute, one must follow the following syntax for reading the variable: `$myvariable`.

```
<xsd:element name="mcbpc" type ="rvc:ext"
rvc:port="Algo_VLDtableB7_MPEG4part2">
    <xsd:annotation> <xsd:appinfo>
        <bs2x:variable name = "mb_type" value = "bitand(./text(),7)"/>
    </xsd:appinfo> </xsd:annotation>
</xsd:element>
...
<xsd:group ref="motion_vector" bs2:nOccurs="4" bs2:if="$mb_type=2"/>
```

C.4.3.6 xsd:simpleType

This element is used to define a new type of `xsd:element` element. The cases in which a new type must 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` children 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.
- To test the value … with an `xsd:startcode` element, see C.4.3.12.

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

Remark: to see different examples of definition of a new type, refer to C.4.3.10, C.4.3.11, C.4.3.12 and C.4.3.14.

C.4.3.7 xsd:annotation

BSDL-2 introduces a set of new facets to specify constraints on BSDL and XML Schema data types. Since XML Schema does not allow a user to add his own facets, they are declared as BSDL-2 components added to the `xsd:restriction` component via the annotation mechanism, i.e. the `xsd:annotation/xsd:appinfo` combination. Thus, the BSDL-2 elements (Table C.4) must be placed as the child of an `xsd:annotation/xsd:appinfo` combination.

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

C.4.3.8 xsd:appinfo

According to the parent element in which this element is called, there are several possibilities:

- If the top parent element is an `xsd:simpleType` element, one can choose to define this new type by using one of the following elements: `bs2:bitLength, bs2:ifUnion, bs2:startcode,` or `bs2:length.`

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

- If the top parent element is an `xsd:element` element of any type except `rvc:ext`, the unique element which can be used is the `bs2:variable` element, used to save variables.

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

- If the top parent element is an `xsd:element` element of type `rvc:ext`, the `bs1:script` element must appear to define which algorithm is used to decode the segment of bitstream. It can be possible also to save a variable using the `bs2:` variable element.

```
<xsd:appinfo>
  Children: {bs1:script, bs2:variable}
</xsd:appinfo>
```

C.4.3.9 xsd:restriction

This element is used to specify data accuracy.

```
<xsd:restriction
  base = "(xsd:unsignedshort | bs1:b1 - bs1:b32 | bs1:byteRange)"
>
Children: xsd:annotation
</xsd:restriction>
```

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

C.4.3.10 bs2:bitLength

This element specifies the size in bits of the current element, which has been defined as a new type using the `xsd:simpleType` construct. The size in bits of the current element can be stored in a variable, which has been assigned during the parsing process.

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

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

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

C.4.3.11 bs2:length

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

```
<bs2:length
   value = "Integer"
>
Children: none
</bs2:length>
```

Example: the `StartCodeType` type instantiates elements of size equals to 4 bytes.

```
<xsd:simpleType name="StartCodeType">
    <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="4"/>
    </xsd:restriction>
  </xsd:simpleType>
```

C.4.3.12 bs2:startCode

This element is used to mark the beginning of the bitstream.

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

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

Example: the type `rbspType` instantiates ….

```
<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.4.3.13 xsd:union

This element 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.

```
<xsd:union
  memberTypes = {bs1:b1 - bs1:b32}
>
Children: xsd:annotation
</xsd:union>
```

Remark: see C.4.3.14 for an example of application.

C.4.3.14 bs2:ifUnion

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

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

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.4.4 Syntax of the expressions

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

```
Expression  →  PrimaryExpression {Operator PrimaryExpression}

PrimaryExpression  →  'max('Expression',' Expression')'
                      | 'min('Expression',' Expression')'
                      | 'numbits('Expression')'
                      | 'bitand('Expression','Expression')'
                      | 'bitor('Expression','Expression')'
                      | 'bitnot('Expression')'
                      | 'rshift('Expression')'
                      | 'lshift('Expression')'
                      | '.text( )'
                      | ExpressionLiteral
                      | ifExpression
```

ExpressionLiteral → '| IntegerLiteral | true | false

ifExpression → if Expression then Expression else Expression end

Operator → (' = ' |' < ' | ' > ' | ' >= ' | ' <= ' | '! = ' | 'and' | 'or' |'not' | '* '| '/' | ' + ' | '- ' | '^' | 'div' | 'mod')

C.4.5 Syntax of the data types

This Subclause describes the syntax of the data types.

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')

Portname → NormativeFUName

The NormativeFUName is the name of the Functional Unit. The naming convention rule is described in ISO/IEC 23002-4.

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

The Syntax Parser and the network of FU must be connected together. Thus, a communication scheme between the syntax parser and Functional Unit is necessary. The following code shows an example of BSD, illustrating the connection of the Syntax parser to an FU.

```
<xsd:element name="horizontal_mv_data" bs0:variable="true" type="rvc:ext"
rvc:port="Algo_mv_reconstruction--mvin"/>
```

The element name "horizontal_mv_data" is decoded by an FU: it is indicated by the data type rvc:ext. The FU which will decode this element of syntax is specified in the rvc:port attribute. The port of the destination FU to which this element of syntax will be sent is specified also in the rvc:port attribute. The name in the rvc:port attribute must correspond to the normative name of the FU to which it is connected in order to know how to make the connections between the syntax parser and the FU network.

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

- 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 bs0: variable 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.

- General output port: an output port which name is value of the rvc:port attribute (e.g. algo_mv). This port is always created. It is used to send the data to the FU.

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:

1) The parser sends data on the port algo_mv

2) The FU receives the data and warns the parser (though the algo_mv_f port)

   i) if it needs an other data (value of the data to return to the parser = false), goto 1
   ii) or if it has finished (value of the data to return to the parser = true), goto 3

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

The example of VLD decoding process using such communication scheme is shown in Annex E.

Annex D
(normative)

Specification of RVC-CAL language

D.1 Generalities

The CAL language is a dataflow-oriented language that has been developed as a subproject of the Ptolemy project at the University of California at Berkeley. The final CAL language specification has been released in December 2003. The specification provided in this annex is the sub-set of CAL language called RVC-CAL used in the MPEG RVC framework. The sub-set 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" that 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:

- The RVC-CAL used to specify the behavior of the FUs that constitute the RVC video tool library
- The 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.

At any point in time, an FU is either disabled, i.e. it is not able to make a step, or it can perform a number of different steps.

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,
- 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 Introduction

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

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 mile-stones [6], [9], [3], [4], [5]. A formal description of the notion of actor underlying this specification can be found in D.1, which is based on the work in [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.

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

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, cf. [11], [12]. Actor composition inside the actor model, that CAL is based on, has been studied in [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.3 Introductory remarks

D.3.1 Introduction

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

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.11.3 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.11.2 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 | OctalLiteral
               Real → DecimalDigit { DecimalDigit } '.' {DecimalDigit} [ Exponent ]
                      | '.' DecimalDigit { DecimalDigit } [ Exponent ]
                      | DecimalDigit { DecimalDigit } Exponent
     DecimalLiteral → NonZeroDecimalDigit { DecimalDigit }
 HexadecimalLiteral → '0' ( 'x' | 'X' ) HexadecimalDigit { HexadecimalDigit }
       OctalLiteral → '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'
```

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 `monospaced`. Semantic entities, such as types, are set *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 $A \mid B$.

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-

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

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.8.2).

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 descriptions

Each actor description defines a named kind of actor.

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

Actor → **actor** ID '(' ActorPars ')' IOSig ':'
{VarDecl}
{ Action | InitializationAction }
[ActionSchedule]
{ PriorityBlock }
end

ActorPar → Type ID ['=' Expression]

IOSig → [PortDecls] '==>' [PortDecls]

PortDecl → 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.6.2), actions (D.9), initialization actions (D.9.6), priority blocks (D.10.4), and at most one action schedule (D.10.3).

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.5 Data types

D.5.1 Introduction

RVC-CAL is fully typed, i.e. it allows and forces programmers to give each newly introduced identifier a type (see D.6.2 for more details on declaring variables).

D.5.2 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 we assume that there is a set T_V of variable types and T_O of object types. They are related to each other by an assignability relation, $\leftarrow \subset T_V \times T_O$, which has the following interpretation: for any variable type t_V and object type t_O, $t_V \leftarrow t_O$ iff an object of type t_O 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 must be compatible with subtyping in the following sense. For any two variable types t_V and t'_V, and any object type t_O:

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

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

D.5.3 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, values are bound using the "=" syntax.

D.5.4 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 Variables

D.6.1 Introduction

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 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.6.2 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.6.2),
- actor parameters (D.4),
- input patterns (D.9.2).

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

D.6.2.1 Explicit variable declarations

Syntactically, an explicit variable declaration[3] looks 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 → [Type] ID [('=' | ':=') Expression]';'
    | FunDecl | ProcDecl

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 way are called stateful variables because they may be changed by the execution of a statement. Variables declared in the last way 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.6.3)

D.6.3 Function and procedure declaration

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

FormalPars → Type ID {',' Type ID}

FuncDecl → **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.6.4 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 e.g. 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 — e.g. 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.7 Expressions

D.7.1 Introduction

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 $\rightarrow$ Expression { ',' Expression }

Expression $\rightarrow$ BinaryExpression
| *S*impleExpression

SingleExpression $\rightarrow$ Operator Expression
| ListComprehension
| ifExpression
| LetExpression
| '(' Expression ')'
| IndexerExpr
| ID '(' Expressions ')'
| ID
| ExpressionLiteral

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

D.7.2 Literals

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

ExpressionLiteral $\rightarrow$ IntegerLiteral | DecimalFractionLiteral
| StringLiteral
| **true** | **false**

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

D.7.3 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.7.4 Function application

An expression of the form

$$F(E_1, ..., E_n)$$

is the application of a function to *n* parameters, possibly none. *F* is the name of a function which must 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.7.5 Indexing

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

$$\text{IndexerExpr} \rightarrow \text{ID '[' Expression']'} \left\{ \text{'[' Expressions ']'} \right\}$$

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

D.7.6 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.7.7 Conditional expressions

The simple conditional expression has the following form:

$$\text{IfExpression} \rightarrow \textbf{if} \text{ Expression } \textbf{then} \text{ Expression } \textbf{else} \text{ Expression } \textbf{end}$$

The first sub expression must be of type `bool` and the value of the entire expression is the value of the second sub term if the first evaluated to **true**, and the value of the third sub term otherwise.

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.7.8 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.7.8.1 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.7.8.2 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** Type ID **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 must 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.8 Statements

D.8.1 Introduction

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.8.2 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.8.2.1 Simple assignment

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

The expression on the right-hand side must evaluate to an object of a value compatible with the variable (i.e. its type must be assignable to the declared type of the variable, if any — see D.5.2). 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.8.2.2 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.8.3 Procedure call

Calling a procedure is written as follows:

```
CallStmt → ProcedureSymbol '(' [ Expressions ] ')' ';'
ProcedureSymbol → ID
```

The procedure symbol must be defined in the current context, and the number and types of the argument expressions must 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.8.4 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.8.5 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 must be of type `bool`.

D.8.6 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.8.7 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** Type ID **in** Expression

The basic structure and execution mechanics of the foreach-statement is not unlike that of the comprehensions with generators discussed in D.7.8.2. 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.9 Actions

D.9.1 Introduction

An action in RVC-CAL represents a (often large or even infinite) number of transition of the actor transition system described in D.10. A RVC-CAL actor description can contain any number of actions, including none. The definition of an action includes the following information:

- its input tokens,
- its output tokens,
- the 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:

Action → [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.10.2. 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.10 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 behavior 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.9.5.

D.9.2 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 must 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.9.3 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.6 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.6.

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.9.4 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 must 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 must contain.

Example. (Output expressions). The output expression

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

produces the output tokens 1, 2, 3.