An Intuitive TTCN-3 Data Presentation Format

Roland Gecse and Sarolta Dibuz

Ericsson Hungary, 1037 Budapest, Laborc 1
{Roland.Gecse,Sarolta.Dibuz}@eth.ericsson.se

Abstract. This paper describes the TTCN-3 Data Presentation Format (DPF). DPF is an intuitive graphical notation for representing TTCN-3 Core Language (CL) [1] types and values. The major advantage of using DPF compared to free-text editing is that a DPF implementation ensures a consistent type and template structure by excluding references to unexisting entities while significantly reducing typing work. The result is a shorter test suite development time. DPF covers all excluded parts of Graphical Presentation Format (GFT) [3]. We believe that DPF and GFT together could be the basis for building the ultimate graphical representation of TTCN-3.

1 Introduction

Traditional test suite design starts with the preparation of declarations and constraints parts. This job requires careful analysis of SUT and good understanding of the exact purpose of the test. A load test, for instance, requires less detailed resolution of PDUs as conformance test. The former focuses on whether the SUT can service a huge number of typical requests while the latter targets to prove standards conformity. Another important factor of data part development is the time. Working out type and template definitions takes 20-50% of total test suite development time. The preparation of data part in CL comprises of free-text editing of textual definitions. Unfortunately, existing presentation formats (TFT [2], GFT [3]) do not provide help on this. TFT basically fragments CL code into table fields while GFT targets dynamic behaviour only. The intention with DPF is to facilitate test suite data part development and thereby make test designers' work easier.

A TTCN-3 module can be subdivided by functionality into several sections. Some of these – constant declarations, type definitions, signature declarations, template declarations, module parameter declarations, port and component type definitions – constitute what we refer to as *data part*. DPF operates on data part only – that is where its name comes from. Other sections containing test cases, functions, alt steps as well as the module control part pass through DPF unmodified (Figure 1).

DPF manifests itself in a utility, which provides a graphical representation of data part constructions and assists the user to manipulate these. This GUI provides separate panes for each section of the data part. The sections have been introduced because we deliberately avoided using a distinct graphical symbol for

	Type	Constant	Signature	Template	Port	Component	ModulePar	Function	Altstep	Testcase	Control
Types	✓		✓		✓	✓					
Templates		✓		✓							
Main		✓		✓			✓	✗	✗	✗	✗

Fig. 1. Mapping of DPF and TTCN-3 parts

each type in order to keep the notation intuitve. This approach resulted that the DPF notation became context sensitive; the same graphical construct can mean different things when appearing at different sections, i.e. a template declaration for a given type, for instance, looks similar to a constant declaration for the same type provided the template consists of specific values only.

The example of Figure 1 shows an imaginary test suite, splitted up into three modules. This example presents how a DPF utility handles TTCN-3 modularity. Each module contains some sections denoted either with a check-mark or a cross depending on whether they are relevant to DPF or not. The module *Main* is the root module of the test suite as it contains the control part governing the test execution. This module includes not only test cases, functions, alt steps (the dynamic part) and module parameter definitions but some additional constants and templates, too. The other two modules appear because the root module imports some or all of their definitions. The module *Types* contains type, communication port and component type definitions and some signature declarations while the module *Templates* holds constants and templates for the types defined in either of the modules.

The GUI provides means for creating or editing adequate entities in every section. A new template can be built, for instance, based on an existing type definition simply by choosing the appropriate type from the type definitions section and assigning values or wildcards to each of its fields. Alternatively, a new type can be defined by extending or reducing an existing type or by merging several types. It could be ensured that no undefined types or templates are referenced, too. Optionally a DPF implementation could perform some basic checks such as if all fields of a template had a value assigned or whether the designer really wanted to create optional set-of types, etc. The fabricated definitions can be saved into a CL module at any time during editing. Regardless of the changes made to the data part DPF keeps dynamic module parts untouched. The only difference between the original CL module and the one that has been processed by DPF is that DPF may reorder some sections.

The rest of the paper presents graphical notations of DPF. The simple type and value notations are described first. It is followed by the structured type constructs. A dedicated section describes the embedded type definition as DPF relieves the TTCN-3 limitation of embedded type definitions. Finally the notation for special CL features like signatures, templates, communication port and component types are introduced. Each section contains some examples together with their CL mapping when approprite in order to get an impression of DPF.

2 Simple Type and Value

The basis of DPF is the graphical notation for simple type as all structured types are *built*[1] of simple types. Figure 2 presents the format of DPF simple type. The graphical representation consists of a rectangle with a mandatory simple type identifier in the middle. Three placeholders are also included for the optional *field identifier*, *with-attributes* and *value* attributes.

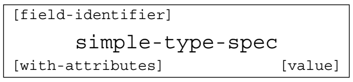


Fig. 2. DPF notation of simple type

The simple types of DPF include all predefined CL simple basic types and basic string types. DPF supports subtyping according to CL 6.2 in [1]. Although the permitted subtyping methods are equivalent DPF subtyping syntax slightly diverges from CL. The mandatory `simple-type-spec` carries all type properties including subtyping. The three optional attributes add further information to the simple type. These include contextual properties, encoding directives and assigned values.

2.1 Simple Type Specification

It has the following ABNF [4] syntax:

```
simple-type-spec =
    [ simple-type-id ":" ] base-simple-type-id [ subtype-spec ]
```

where `simple-type-id` and `base-simple-type-id` shall be unique identifiers within the scope of the module. `subtype-spec` is any valid CL subtype specification (`SubTypeSpec`, prod. 44 of CL BNF) of the corresponding `base-simple-type-id`.

Depending on presence of optional parts `simple-type-spec` can be a *simple type reference*, a *simple type alias*, a *subtype definition* or an *inline subtype definition.*

[1] The actual building methods are introduced at the structured types.

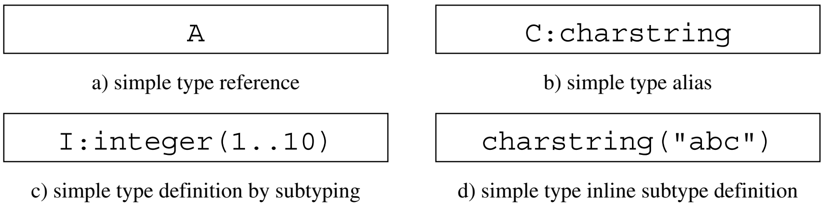


Fig. 3. DPF notations of simple type constructions

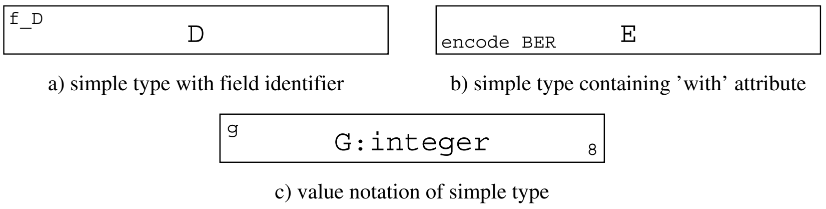

f_D
D
a) simple type with field identifier
encode BER
E
b) simple type containing 'with' attribute
g
G:integer
8
c) value notation of simple type

Fig. 4. Examples of simple types with attributes

1. The simple type reference (Figure 3/a) consists of a reference to a predefined simple type or a user-defined simple type. Both optional `simple-type-id` and `subtype-spec` are absent.
2. Simple type alias (Figure 3/b) is composed of a unique simple type identifier followed by a semicolon and the reference to an existing simple type. This construct is used to create a new simple type (an alias) with the given identifier from a base type. The new alias type has the same value domain as its base type. The CL mapping of the example is `type charstring C;`.
3. The subtype definition in Figure 3/c constructs a new simple type from an existing simple type by means of permitted subtyping methods of `base-simple-type-id`. The resulting subtype is considered as a distinct simple type. The CL equivalent of the example is: `type integer I (1..10);`.
4. The inline subtype definition (Figure 3/d) derives an unidentified new type from another simple type using subtyping. The difference between subtype definition and inline subtype definition is that the latter can not be referenced because it has no identifier.

It shall be noted that items 1 and 4 may occur in structured type definitions only or additional attributes must be added to provide semantics!

2.2 Simple Type Attributes

The DPF representation provides three attributes to describe the context of simple type specifications. The following paragraphs introduce each in detail.

Field Identifier. The optional context specific `field-identifier` (Figure 4/a) has different meaning in different module parts. It can stand for the identifier of a constant or variable of the current simple type or the field identifier assigned to a structured type element.

CL requires an identifier to be assigned to each structured type element in order to distinguish between them. DPF provides graphical symbols for this purpose that are sufficient in most cases. Consequently identifiers can become superfluous and may be omitted. However, the identifiers must be provided when exporting definitions into CL. Thus, missing field identifiers are automatically generated during the export[2]. The `field-identifier` shall be unique within the scope of the given structured type.

With Attributes. The `with-attributes` field (Figure 4/b) is the location of CL with attributes. The `with-attributes` can be separately set for each simple type in the bottom left corner of the rectangle symbolizing the simple type. The syntax is similar to the format of CL with statement (CL BNF prod.491) except that `WithKeyWord` and the enclosing curly braces are omitted. The content of the `with-attributes` field shall be inserted into the with attributes of the given simple type when exporting DPF into CL.

The Value. The bottom right value field is only used in value notation and must be empty in type definitions. This field represents the value assigned to the type instance specified in field identifier. It holds a specific value or expression, which is valid for the current profile and results in a value adequate to the type defined. The example in Figure 4/c presents a simple type instance `g` of type `G`, an `integer` alias, having the value 8 assigned.

3 Structured Type and Value

DPF follows a general technique for building structured types; it defines operations for constructing structured types. These operations are ordering, choice, permutation and repetition. Ordering, which joins types into a sequence is expressed by the record construct. Choice offers type alternatives equivalent to union type of CL. Permutation of element types is represented with unordered set. Finally, type repetition is modeled with quantors.

DPF type definitions are represented on a plane where the imaginary vertical axis specifies ordering while the horizontal shows the potential alternatives. The axes themselves are invisible in DPF. The representation can always read unambiguously in top-down and left-right direction.

The graphical notation of structured types (Figure 5) consists of a tag incorporating the list of element types arranged on the plane according to the introduced principle.

[2] A suitable procedure is to generate `field-identifier`s from type identifiers by prefixing with literal `f_` and postfixing with `_n`, where `n` is the index of occurrences of the given type within the given context.

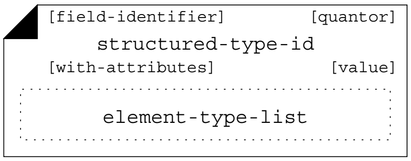


Fig. 5. DPF notation of structured type

The tag contains the structured type identifier (`structured-type-id`) and optionally the `field-identifier`, `quantor`, `with-attributes` and `value`. The meaning of these is similar to the identically named properties of simple types.

Field Identifier. The optional `field-identifier` can be used to assign an identifier to an instance of an embedded type definition. In value notation it is used to identify the appropriate constant, variable or template instance.

Quantor. The quantor is used to express record-of and set-of constructs (see section 4.5) including subtyping facility. The quantor is placed into the upper right corner of the tag. It can only be used when the construct comprises of a single element type! The quantor must appear in value notation but it must be omitted in stuctured type reference. The quantor has the format (ABNF):

```
quantor   =   min [ ".." max ]   /   "*"   /   "+"
```

where `min` and `max` are integral numbers such that `max>min`. The shorthands `"*" = 0..infinity` and `"+" = 1..infinity` are also allowed.

With Attributes. The optional `with-attributes` has the same syntax and semantics as the with attributes of simple types in section 2 except that the with attributes specified for structured type apply for all element types. In case of an element type reference appearing in `element-type-list` has different with attributes this overrides the with attributes of the structured type.

Value. The value field shall only be used in value notation. It may hold either a reference to a structured type value or an expression that evaluates to the given structured type value. The value specified in the tag can be partially superseded by values defined in element list.

3.1 Element Type List

The `element-type-list` determines structured type content. This can include references to simple or complex types as well as embedded type definitions. The graphical format and layout of `element-type-list` depends on the construction used for structured type definition. Section 4 presents the graphical format of each structured type in detail.

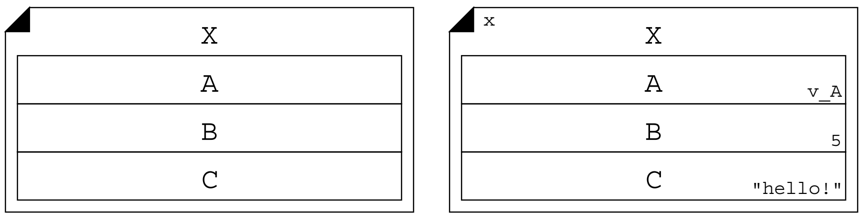


Fig. 6. Example for a record type and value notation

The `element-type-list` is omitted at structured type reference and structured type alias constructions. The graphical notation of these is identical to their counterparts at the simple types, i.e. simple type reference as shown in Figure 3/a and simple type alias in Figure 3/b.

4 Structured Type Constructions

4.1 Record Type and Value

Fixed ordering of element types is expressed using the record construction. The graphical notation of record type is a chain of element types along the position axis arranged such that rectangles of neighboring element types have joint edges. The record type definition example on Figure 6 is shown together with its value notation. Type `X` consists of an ordered list of types `A`, `B` and `C`. None of the element types have field names assigned. Note that `A`, `B` and `C` are not necessarily simple types. The value notation to the right assigns `A` the value `v_A` by reference, while `B` and `C` get the literals `5` and "`hello!`" respectively. The equivalent CL definitions are:

```
type record X {            const X x := {
    A   f_A,                   f_A := v_A,
    B   f_B,                   f_B := 5,
    C   f_C                    f_C := "hello!"
}                          }
```

4.2 Union Type and Value

The union construct expresses a choice of alternative types. DPF represents type alternatives orthogonal to the position axis. The element types are enlisted similar to the record construct but along the horizontal axis. The example in Figure 7 (left) defines the union `Y` as an alternative of the three element types `A`, `B` and `C`.

The value notation of union type permits only a single alternative to get value assigned. The not selected alternatives can as well be omitted. The example in Figure 7 (right) shows a union value containing element type `B` having value `5` assigned. The CL equivalent definitions are:

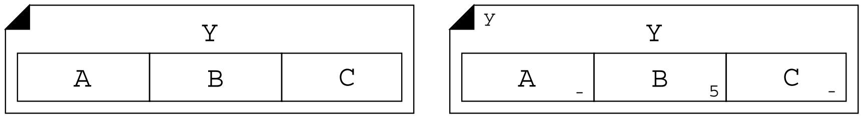


Fig. 7. Example for union type and value

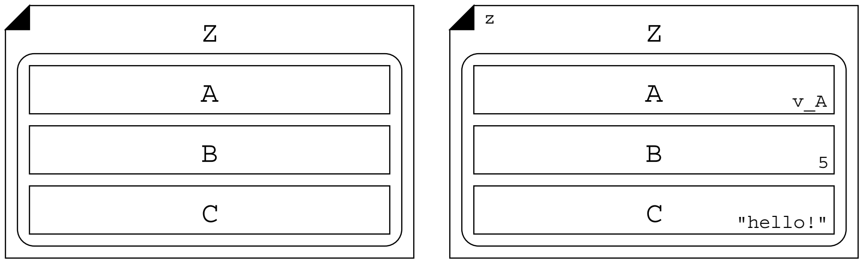


Fig. 8. DPF notation for set type and value

```
type union Y {          const Y y := {
    A   f_A,                f_B := 5
    B   f_B,            }
    C   f_C  }
```

4.3 Set Type and Value

Arbitrary ordering of types is expressed with the set construct. The graphical representation of set is formed by enclosing the element types into a rectangular box with rounded corners such that the element types must not touch each other.

The example set type `Z` in Figure 8 (left) stands for any permutations of element types `A`, `B` and `C`. The value notation for the set type (Figure 8 (right)) assigns value to each element type either literally or by value reference. The equivalent CL definitions are:

```
type set Z {           const Z z := {
    A   f_A,               f_A := v_a,
    B   f_B,               f_B := 5,
    C   f_C                f_C := "hello!"
}                      }
```

4.4 Optional Elements in Structured Types

Protocol specifications often make use of optional types. Optional types may occur in record and set types. The graphical notation distinguishes optional element types with a dotted border. The element types `B` and `C` in the example in Figure 9 are optional in both `XO` record and `ZO` set types.

The CL equivalent type dinfitions are:

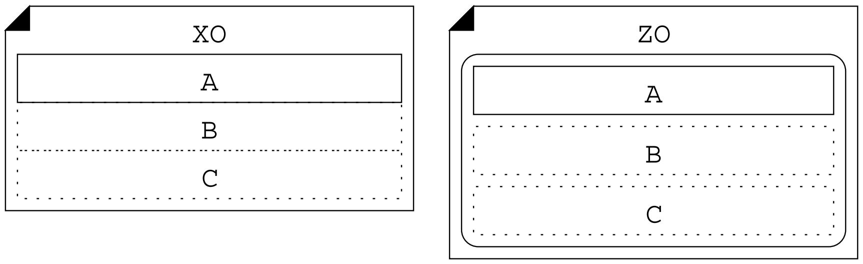


Fig. 9. DPF notation for optional element types in record and set types

```
type record XO {              type set ZO {
    A   f_A,                      A   f_A,
    B   f_B optional,             B   f_B optional,
    C   f_C optional              C   f_C optional
}                             }
```

The value notation of structured types containing optional element types is similar to structured types except that omitted optional element types get the "–" (omit) value assigned. Figure 10 contains example values for both types defined in Figure 9.

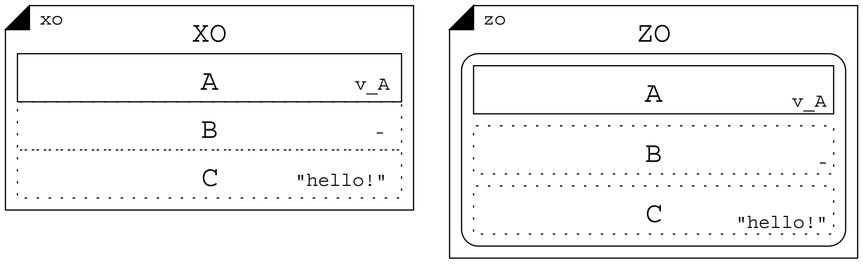


Fig. 10. DPF value notation for optional record and set types

The CL mapping of the constants in Figure 10:

```
const XO xo := {                const ZO zo := {
    f_A := v_A,                     f_A := v_A,
    f_B := omit,                    f_B := omit,
    f_C := "hello!"                 f_C := "hello!"
}                               }
```

4.5 Record and Set of Types

The quantor inside the tag of record or set types expresses record-of or set-of types. The element type list of set or record types definition must consist of a single element type. Note that the quantor can put length restrictions on set-of and record-of types and thereby express subtype constraint. The set-of type

P of the example in Figure 11 stands for zero or more unordered appearances of element type A. The record-of type R represents an ordered list of 1 to 3 repetitions of type A.

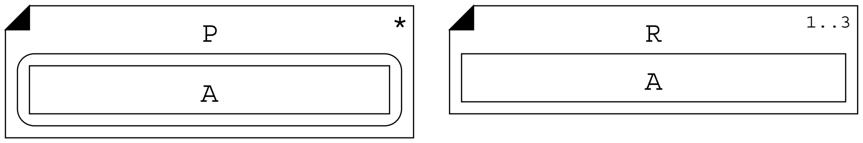


Fig. 11. ITN notation for set-of and record-of types

The equivalent CL definitions are:

```
type set of A P;              type record length(1..3) of A R;
```

The type notation for set-of and record-of types shall contain as many instances of its element type as specified by the quantor. The value of the quantor must always be in accordance with the type definition. The set-of and record-of values of the example in Figure 12 both consist of three element instances.

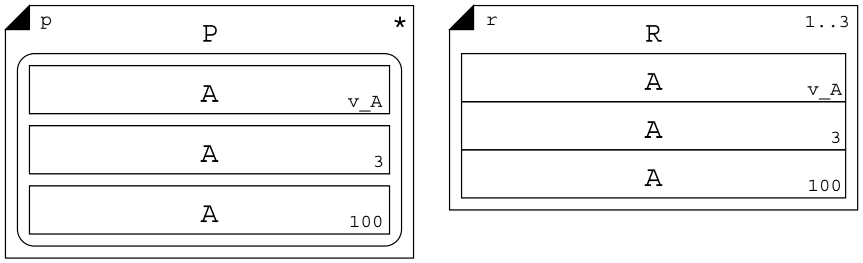


Fig. 12. Value notation for set-of and record-of types

The corresponding CL declarations are:

```
const P p := { v_a, 3, 100 }; const R r := { v_a, 3, 100 };
```

5 Embedded Type Definitions

TTCN-3 forbids embedded type definitions. DPF on the contrary provides graphical notation for embedded type definitions. The only requirement is that even the embedded type shall have an identifier assigned. The `element-type-list` part of structured type definitions may contain embedded type definitions of both simple and structured types. At embedded typed definition, the type definition to be embedded simply replaces its reference in the element type list.

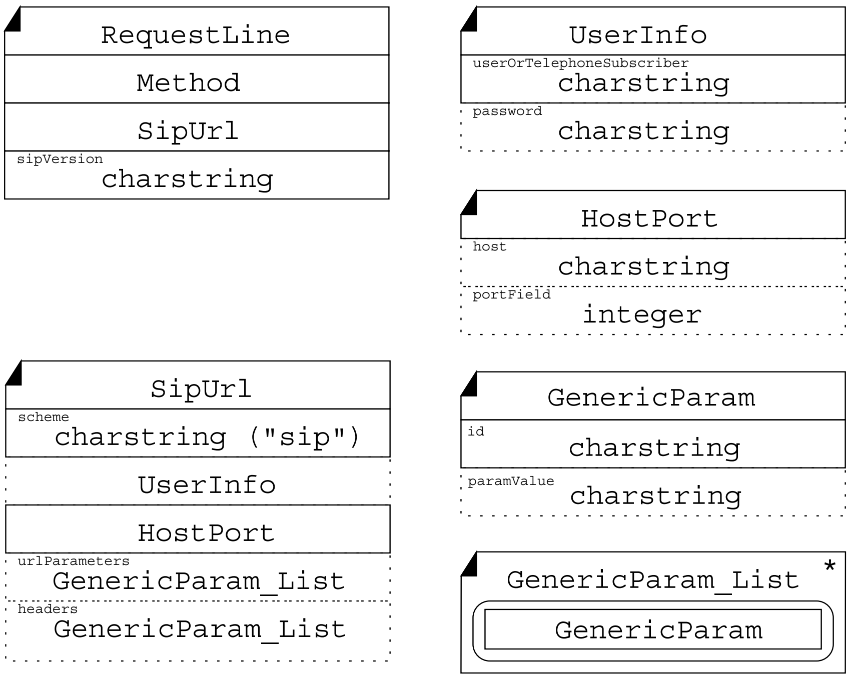


Fig. 13. DPF “flat” definitions for SIP RequestLine

The inner type inherits the field name and value attributes of the reference. The frame format (optional, mandatory) is also kept in the embedded type. When exporting DPF definitions into CL format, all these definitions shall be decomposed into standalone “flat” type definitions.

DPF implementation shall enable editing in flat view (Figure 13), in embedded view (Figure 14) and in mixed view (Figure 16). Furthermore, it shall support conversion between these modes by expanding references into embedded type definitions or vice versa collapsing embedded type definitions into references. The default type view could be the uppermost level flat view. The desired type reference shall then be expanded into an embedded definition at user request. The embedded view of a type could be collapsed when the user finished editing. This way the user could keep overview of the whole type hierarchy.

The example in Figure 13 shows structured type definitions of RequestLine part of the SIP Invite-Request message. The complete definition consists of five tables (except the simple type definitions). The equivalent embedded definition in Figure 14 fits into one table. Although the embedded notation can be harder to read in this printed form it expresses element type hierarchy and gives a better overview of the entire complex type definition.

The embedded graphical representation increases overview of complex data definitions and thereby helps to detect certain unconformities, which may occur in definitions. These include record-of/set-of types referenced in an optional

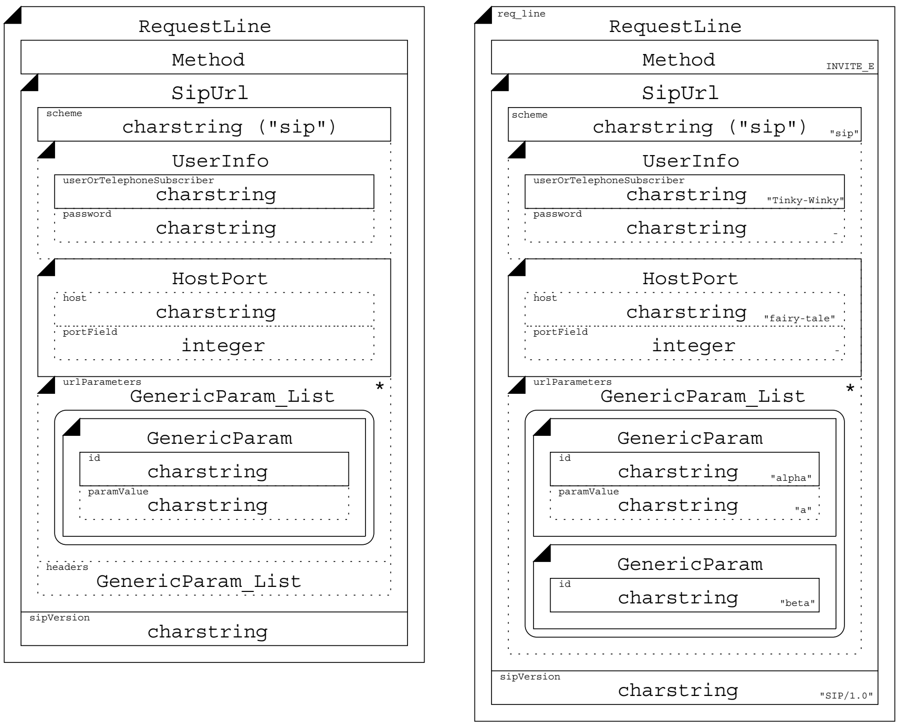


Fig. 14. Embedded DPF view of SIP RequestLine with an example value

field, an optional definition of a record/set consisting of solely optional elements and much more. The GenericParam_List type, which is defined to be a set-of GenericParam type, for instance, appears twice as an optional field of SipUrl. This feature can further contribute to correctly designed declaration parts.

The value notation for embedded type follows the same way as ordinary structured types. Figure 14 (right) presents a dummy value for RequestLine.

6 Recursive Definition

The recursive definition is a special case of embedded definition. The defined type contains a reference to itself. Following the CL guideline DPF does not place any restrictions on recursions therefore it is the user's responsibility to make sure that no endless recursion loop is defined. Figure 15 presents a simple example recursive type definition (left).

```
type record U {           var U u := { f_a := 1, f_U := {
    a   f_a,                           f_a := 3, f_U := {
    U   f_U optional                   f_a := 5, f_U := omit } } };
}
```

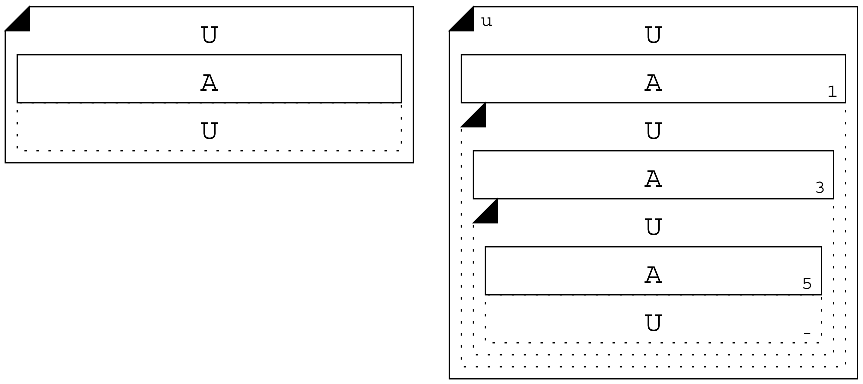


Fig. 15. Example of recursive type definition and value notation

The value notation, however, must not contain recursion. The recursive definitions shall be unfolded similarly to embedded type definitions before getting a value assigned. A matching value for the example recursive type definition is shown in Figure 15 (right).

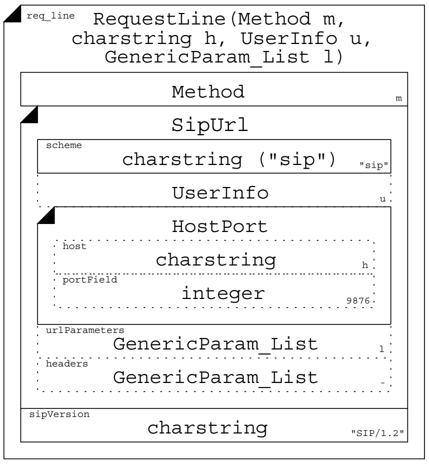
7 Signature Declaration

The declaration of signatures is a prerequisite of procedure based communication. In DPF, the signature declarations are collected into a separate section. The graphical representation of a signature resembles to structured type definition (Figure 5). The signature name takes the place of structured type identifier. The return type identifier replaces the field identifier while the exceptions go into the top right corner into the position of the quantor. In case of non-blocking signature types the `noblock` keyword replaces the return type in the top left corner of the tag. The signature parameters appear on the element type list in an ordered fashion. For each item of the parameter list: the parameter type goes into the middle, the parameter name into the top left identifier while the direction indicator keyword into the top right corner. Figure 16 (right) contains the DPF equivalent of the following CL signature declaration:

```
signature IPConnection (inout ProtocolID protocol,
  in IPAddress srcHost, in integer srcPort,
  out IPAddress dstHost, out integer dstPort)
return SuccessIndication exception (ErrorException);
```

8 Template Definition

DPF provides graphical representation for CL templates. The notation resembles to value notation. The major difference is that wildcard symbols expressing



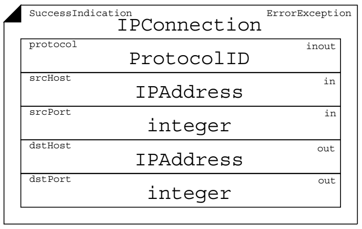


Fig. 16. DPF example for parameterized template definition (left) and signature declaration (right)

matching mechanisms can be used beside specific values and value references inside the bottom right value attribute. Parameterized templates are also available. The formal parameters list must be specified in parentheses behind the template identifier inside the tag. The formal parameters can then be incorporated as references in expressions of the value field.

DPF also supports modified templates. The name of the base template shall appear in the value attribute of the outermost template definition. The GUI shall then fill all elements according to the base template and offer the user to make changes. The implementation keeps track of the changes and creates appropriate modified templates when the module is exported to CL. A benefit of this approach is that the user always see the full template even if she works with modified templates. Optionally DPF implementation could also perform template optimization, which includes the generation of modified and parameterized templates when feasable.

Note that the representation on Figure 16 (left) contains both embedded definitions and references at the same time. This is an example of a mixed view.

9 Component and Port Type Definitions

Component and port types belong to different DPF sections. Both are based upon the graphical notation of the set type (Section 4.3).

DPF supports procedural, message based as well as mixed port types. The `structured-type-spec` of port type definition contains the name of the given port type. The field identifier consists of the literal `procedure`, `message` or `mixed` signalling the operation method of the port. The element type list consists of messages, procedure signatures or both consequently. The message type defini-

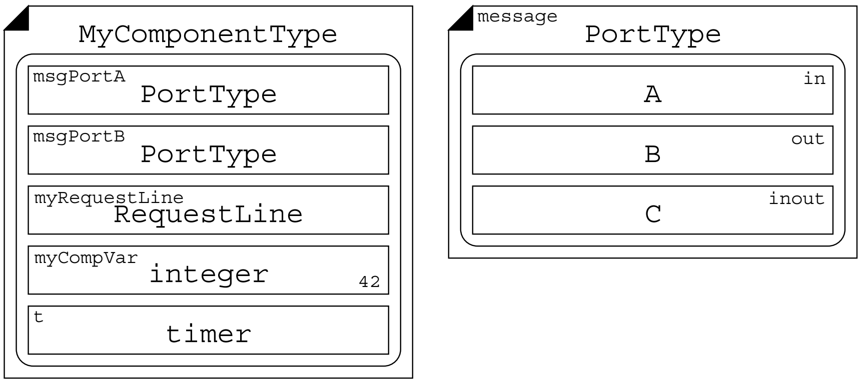


Fig. 17. DPF notation of component and port type definitions

tions and procedure signature declarations come from their respective sections. The keyword `in`, `out` or `inout` appearing on the top right corner determines the direction of the message or signature. Figure 17 (right) shows an example port type definition.

The graphical layout of component type is similar. The component type name belongs to the `structured-type-spec`. The component type definition must not have any attributes. TTCN-3 components may contain port, timer and variable declarations. The element type list uses DPF value notation. Port declarations consist of a reference to a port type and a port name inside the field identifier. Ports must not have other attributes that the field identifier. Variables are also declared by reference but these may have an initializer present inside the value attribute. Timers are declared using the `timer` keyword as `simple-type-spec`. Similarly to variables, timer may also have an initial value set. The example in Figure 17 (left) shows the DPF definition of MyComponentType. The CL equivalent definitions:

```
type component MyComponentType {       type port PortType message
    port PortType msgPortA, msgPortB;  {
    var integer myCompVar := 42;           in    A;
    var RequestLine myRequestLine;         out   B;
    timer  t;                              inout C;
}                                      }
```

10 Conclusion

We presented a new TTCN-3 presentation format, the Data Presentation Format, for intuitive graphical representation of TTCN-3 data. DPF provides graphical notation for all kinds of CL types and values, which is especially useful in large-scale test development where protocol(s) contain several hundreds or thousands of type definitions and templates. A typical DPF implementation enhances test data design and thereby reduces the lead time of test suite development. We hope that DPF gets acceptance from the testing community either as a stand-alone presentation format or as a complementary part of GFT.

References

1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, ETSI ES 201 873-1 V2.2.0 (2002-03).
2. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT), ETSI ES 201 873-2 V2.2.0, (2002-03).
3. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT), ETSI TR 101 873-3 V1.2.1, (2002-05).
4. D. Crocker, Ed., P. Overell: Augmented BNF for Syntax Specifications: ABNF, RFC-2234, Nov. 1997.