Eclipse/ExDT – Tools For Interactive Development Environments Setup

Table of Contents

Introduction

This document describes general principles of the tools for interactive development environments (IDE) custom setup, being developed by Excelsior, LLC for Elphel, Inc., which are aimed at various technologies of hardware design, but first of all, based on FPGA and Verilog.

Requirements

On of the motives for starting these works was Elphel's need for convenient hardware developer's tools. The experience of using some existing tools had discovered lack of universal technologies and irreproachable tools in this field and also their insufficient customizability, compatibility and combinability with each other. The existing programming means most commonly are closed packages of ready-to-use solutions provided by hardware vendors, which are primarily oriented to their technologies and do not allow integration with other tools at all or allow it with significant efforts. As for alternative tools, it's often that they cover only single phases of the whole technology (e.g. Verilog simulation), and secondly, they are controlled either by command line languages that require writing complicated scripts, or within a specific interactive shell. In any case, if a developer uses a unique technology of his own, which parts are supported by different instruments, he has no chance to run all phases of this technology within one development environment and makes significant efforts to create huge OS command line scripts that automate his special development procedures.

Therefore the development environment to be created within this project has to meet the following requirements:

- *extensibility* – ability to integrate as more existing or new programming means as possible;
- *polyvariancy* – ability to set up several technological procedures to be available at the same time, also implemented with alternative instruments;
- *flexibility* – ability to change setup settings within a certain installation or for a certain user;
- *simplicity* – ability to perform the above works by an end-user (hardware developer) himself;
- *portablity* – ability to port a technology to various operational platforms, in particular, Windows and Linux.

All these features are hereafter considered as aspects of the ***custom setup of an IDE***.

The extensible IDE *Eclipse* has been chosen as an implementation platform.

Eclipse

Eclipse is an open IDE, which can be customized for a wide range of programmatic development technologies. There exist many programming systems based on Eclipse and specialized for well-known language technologies (C++, Java, etc.) and application packages.

Eclipse can be customized programmatically, by means of extensions ***(plug-ins)*** programmed in Java, that add new features to the IDE by providing implementations of predefined interfaces of various functional elements of Eclipse. Via interfaces, plug-ins can also gain access to other features of the IDE.

Opportunities of programmatic customization are quite wide; however, even despite the openness of the IDE, developing a plug-in for Eclipse is a laborious work for a qualified and

experienced programmer, not for an end-user. This is also true for customizing ready plug-ins, for example, when adding new tools into the IDE or customizing existing tools is needed.

Eclipse/VDT

The target, end-user variant of a customizable IDE, oriented for technologies on base of the Verilog language and meeting the above requirements, is being developed under the name of **Verilog Development Tools plug-in for Eclipse** (**Eclipse/VDT**).

The end-user description of this plug-in may be found in the *«Eclipse/VDT User's Manual»*. It is recommended to get acquainted with this document first to learn more about general features of the plug-in.

Plug-in Eclipse/VDT allows to integrate third-parties' application tools (utilities and packages of utilities) into Eclipse that a user may need for his project development. Hereafter, all actions for integration and operation control of these tools are called ***tools setup***.

Eclipse/VDT provides two levels of tools setup:

- ***operational*** setup of tool's work mode – is performed by means of control elements provided by the IDE for a particular project: the values of configurable parameters may be varied in properties setup dialogs and then passed to utilities in command lines or control files; this kind of setup is typical for interactive applications;
- ***functional*** setup, that is choice of available tools set and description of their control interfaces – is performed by means of an editable XML specification, where defined are all configurable control parameters and their properties, design menus, single tools and packages, appearance of properties setup dialogs, command lines and control files of utilities; this kind of setup is untypical and is the distinctive feature of Eclipse/VDT.

Functional setup is performed ***dynamically***: it requires no reprogramming, retranslation or reinstallation of the plug-in – changes in XML files take effect next time Eclipse starts.

While developing and probing this plug-in, it turned out that opportunities of implemented means for functional setup exceed the bounds of the application field of initial interest: they can be efficiently applied to support a broad range of programming tools development technologies.

That's why the functional setup means have been taken out into a separate plug-in – Eclipse/ExDT – of higher abstraction level and of instrumental purpose, in terms of which they will be described below.

Eclipse/VDT is now considered as Eclipse/ExDT version programmatically specialized for development technologies in Verilog. In future these two plug-ins will be logically separated, so that Eclipse/ExDT will become a general-purposed toolkit for customizable IDEs development , and Eclipse/VDT will turn into just one of its possible programmatic/functional specializations for a paricular application area.

In the rest of this paper the issues of functional setup are discussed as features of Eclipse/ExDT.

Eclipse/ExDT

The **Extensible Development Tools plug-in for Eclipse** (**Eclipse/ExDT**) is being developed as a multi-purposed tool kit for configurable IDEs development with Eclipse.

The opportunities of functional setup in Eclipse/ExDT, implemented by dynamically interpreted XML specification of various properties, lay on an intermediate level between wide opportunities of programmatic setup in Eclipse and limited opportunities of operational setup in Eclipse/VDT and other specialized IDEs.

As against of specialized IDEs, Eclipse/ExDT provides no ready programmatic tools to solve end-user's applied tasks. It contains only an interface shell and a tools kernel, which a user may fill himself with selected applied programs and packages, thus creating a specialized IDE (also easily respecializable).

Possession of a command line control interface is the only requirement limiting integration of this or that program in Eclipse/ExDT.

TSL

Functional setup of Eclipse/ExDT, i.e. specification of programmatic tools and their control elements available to an end-user, is described in an internal ***Tools Specification Language – TSL***, based on XML. The TSL specification is completely open to a user and requires no plug-in recompilation as it is interpreted when loading Eclipse/ExDT.

TSL is a metalanguage which introduces, in fact, three interconnected control interfaces:

- ***user control interface (input)***, related with setup parameters visualization and input;
- ***command-line interface of a program (output)***, related with program's control command language and passing parameters to a program when called;
- ***data representation interface (internal)***, related with internal organization of TSL elements and their relations.

Hereafter, terms and explanations related to each of the above interfaces are hilighted with respective color.

Basic Components Of The Eclipse/ExDT Technology

This section introduces terms and definitions for basic concepts used in Eclipse/ExDT, enumerates basic components of the plug-in and clarifies their purpose.

Projects And Technologies

All user's activities in Eclipse are carried out within a ***work project*** and are related with processing of project data by means of some set of tools. The structure and properties of these tools depend on a project type defined by a plug-in.

Build/design of a project consists in obtaining the required set of output data from the initial set of input data. A project build process may go stepwise, with intermediate data producing. It may be performed either manually, by direct data edition, or automatically, with programmatic tools, or in mixed way.

(Terminology note: in microcircuits production, a project is often called a *design*, and to build it is to *implement a design*; therefore this term is used in Eclipse/VDT).

Build/design flow is a chain of elementary data processing steps implementing some method of project building.

Project build/design technique is the overall set of methods and tools used for project building.

Perspective And Tools

A ***perspective*** defines a set of tools and control means available to a user, appearance of control panels and their arrangement in the Eclipse window.

There are three kinds of tools in Eclipse/ExDT:

- ***navigators***, which visualize the project's contents in various representations ***(views)*** and allow to select the current component to process;
- language-oriented ***editors*** of text files;
- ***application tools***, which call external utilities for automatic data processing.

Data Objects, Navigators And Editors

Basic data objects in Eclipse/ExDT are ***files*** of different types. A data type corresponds to file contents language or format and is detected by its name's extension. A project is visualized as a list of files by the file navigator **File View**.

Various programming languages components may be other data objects. For their visualization special navigators may be programmed. For instance, the **Module View** navigator has been implemented for Eclipse/VDT, which visualizes a project as a hierarchy of Verilog modules.

File editors are associated with files of certain types and may have language-oriented features, such as syntax hilighting, context helps, navigation across a list of error messages etc..

At the moment, navigators and editors for Eclipse/ExDT and its special versions are chosen from those existing in the world or are programmed separately. Their integration is performed programmatically.

Application Tools Menu

There is a hierarchical ***design menu*** in Eclipse/ExDT, where represented by buttons are ***tools*** denoting calls of external utilities with storable parameters.

The term "tool" is associated not with the program itself, but with some kind of service performed by it in some possible operation mode: thus, a tool is an elementary, logically detached step in a design flow, which may be implemented either by a single program or by a sequence of programs united in a batch-file.

The design menu is a configurable element of the current project. It may be built individually for each design technique. Typical menus are supposed to be supplied with tool packages and describe vendor's design techniques, and a programmer may customize them according to the specifics of his project's techniques.

Descriptions of the structure and contents of a design menu are written in TSL and are integrated in the IDE dynamically.

Availability of each tool in the menu at each moment depends on whether its application to the current data object is valid.

For each tool in the menu the following actions may be performed:

- ***operational setup*** – actions to vary control parameters values in properties dialogs;
- ***run*** – building a command line and/or command files for the program associated with this tool, running this program, caption of its output log in a separate console window.

Additionally, the following actions may be performed in the menu:

- ***installation setup*** of the Eclipse/ExDT properties and also of application packages and project properties;
- operational setup of application tools properties, which are common for the package or for the project.

Parameters And Tools Properties Setup Dialogs

User control interface defines the ways of operational tools setup by a user in work time.

Every tool ic controlled by its special set of ***parameters***, for which one of possible ***values*** may be set according to their ***type***. In ***properteis setup dialogs*** current values of available parameters may be modified.

Parameters in setup dialogs may be divided into logical ***groups***. In a setup dialog, a separate ***tab*** corresponds to each group. Every parameter is represented in a dialog by an explanatory ***label*** and an ***input control***. The kind and appearance of an input control depends on a parameter's type and may be one of the following:

- ***check-box*** – for boolean parameters;
- ***editable text box*** – for text and numeric parameters;
- ***drop-down selection list box*** – for enumeration parameters (with fixed sets of values);
- ***browse button*** – for files and directories selection;
- ***list value input dialog*** – for parameters allowing lists of elements as values.

There are several levels of parameters setup (see *«Contexts and Parameters Visibility Rules»*) with a separate setup dialog for each. All setup dialog are available from the design menu.

Utilities And Control Command Languages

A utility is a single program doing the job of one or more tools. Utilities may be ***stand-alone*** or unite in ***packages*** which support some through development technique. Utilities in a package may have common ***location*** (that simplifies their setup) and uniformity of control languages (that simplifies their description).

Any program may be integrated into Eclipse as a utility, if its call can be specified by means of control command interface.

(A program need not necessarily be a console application: it may display graphical dialogs as well – however, no interaction with other Eclipse dialogs is provided for them.)

A control command interface defines ways of calling a utility and passing it control parameters.

When a utility is called, control information is passed to it in one or more ***control lines*** carried by the only ***command line*** and, possibly, one or more ***command files***. The contents and format of control lines is defined by the utility's control language and by the call specification.

A control line is composed as a list of ***options*** with given ***settings***. Every allowed combination of options settings specifies one of possible ***operation modes*** of the utility.

The ***standard output stream (stdout)*** defined for a console application is redirected into a separate ***console window*** of Eclipse to view and process.

Correspondense Between Input Parameters And Output Options

A tool run is performed as a launch of the utility implementing its function; with this, input parameters are converted in utility's options, and parameters values – in options settings. The typical conversion methods ***(output formats)*** may be predefined and then chosen for each parameter individually.

Thus, there is a mapping between tool's parameters and utility's options which is commonly one-to-one. This mapping is defined in TSL by means of a `parameter` object, which combines in one ***the input representation*** of the parameter (how it is shown and input in a setup dialog), its ***output representaion*** (how it is converted in an option setting) and its ***internal representation*** (attributes and relations with other parameters and objects of the TSL specification).

However, not all existing utility's options should be becessarily setup via parameters: some of them may be passed as constant settings fixing some operation mode of the utility which implements the function of the given tool. Also, sometimes absence of an option in a command line may mean passing it with the default value.

Contexts And Parameters Visibility Rules

A definition scope for parameters in Eclipse/ExDT is a ***context*** of operational setup which defines:

- a list of defined parameters;
- contents and appearance of a parameters setup dialog;
- a set and formats of control lines generated for a utility.

There is a hierarchy of 4 kinds of contexts in:

1) ***installation context*** – specifies the basis of "system" settings for all utilities in Eclipse/ExDT;
2) ***package context*** – corresponds to "configuration" settings of a utility in a package;
3) ***project context*** – corresponds to "project" settings of a utility;
4) ***tool context*** – corresponds to a particular utility call.

The contexts had appeared for the reason that there exist programs which parameters differ by generality:

- There exist parameters that define configuration properties of the whole package installation at the given computer (e.g. processor's word length or endianness); typically, their values are the same for all user projects on this machine, but may change when porting to another machine. These parameters are denoted as *package-level* settings. Some programs accept such settings in a special *configuration file*.
- There exist parameters that define properties of a particular project (e.g. architecture of a target platform) and should be preserved when porting the project to another machine. These parameters are denoted as *project-level* settings. Some programs accept such settings in a special *project file*.
- There exist parameters that define an operation mode of a particular utility call (e.g. debugging build mode). These parameters are denoted as *call-level*.

It should be emphasized, that all these parameters control only the utilities in which command language they are defined; all these "project" and "configuration" files composed of them are just special ways of passing parameters to a utility, but in no way properies of a work project of current configuration of Eclipse/ExDT.

- To *installation-level* settings fall parameters that denote other properties than of particular packages, projects or tools: e.g. general properties of Eclipse/ExDT installation on a given machine (OS name, executable files extension in this OS, etc.). It is unlikely that these parameters are known to all utilities, so they are supposed to be used not for immediate

passing to utilities, but in TSL conditional expressions, to select components used with the given configuration of parameters.

The following ***rules of parameters visibility*** in contexts are defined in TSL:

- the rule for parameters ***penetration***: in the above hierarchy of contexts, parameters penetrate upside-down, i.e. a parameter defined in some context is considered known without definition also in all lower contexts by this hierarchy.
- the rule for attributes ***overriding***: is a parameter penetrates in the context which already contains another definition of the same parameter, then only those attributes penetrate which are not specified in the lower context.

For example, if the default compiler's stack size is defined in the configuration file, it may be redefined in the project file and then again in a call's command line.

TSL Basic Concepts And Objects

This section introduces basic concepts, constructive elements and working mechanisms of TSL that are used for functional setup specification in Eclipse/ExDT.

TSL Data Model

The elements of TSL specification are ***objects***, which have:

- ***attributes***, denoting various object properties;
- ***structure***, defining a hierarchy of objects definitions (structural dependency relations);
- ***relations***, defining various other relations among objects (except of structural dependency).

The catalog of existing objects and their hierarchy are:

- *interfaces*
  - *types*
  - *options formats*
- *contexts*
  - *parameters*
  - *input sections*
    - *input groups*
  - *output sections*
    - *control lines*
- *menus*
  - *submenus*
  - *tool calls*

Data in TSL are:

- parameters values with input, output and internal representations;
- control lines and their elements – options with output representations only;

Input *representations* specify how data are displayed and input in setup dialogs.
Output *representations* specify how data are passed to utilites.
Internal *representations* specify how data are stored in Eclipse metadata.

Objects ***attributes*** may be:

- own name (identifier) of an object and names of objects in relation with it;
- parameters values in one of the above three representations;
- explanatory texts in dialogs;
- control lines and options formats;
- special values;

Representations of all objects are **strings** (there are no other data types in XML).

Relations among objects are established by objects names.

Virtual Parameter Types

As mentioned above, all parameter values are represented by string. The ***mechanism of virtual typification of parameters*** in TSL allows to treat parameters as typified, i.e. to limit sets of valid values and to perform values input and values conversion among representations differently.

Associated with each parameter is a TSL `type`-object that defines these limitations. The following basic virtual types are defined:

- ***boolean types***: it is possible to define several types differing by output representations of `true` and `false` values (e.g., `"Yes"` and `"No"` or `"+"` and `"-"`);

- ***numeric types***: allow limiting the range of acceptible values and different ways to format values (e.g., `"12'333.00"`);
- ***string types***: allow limiting a string length, case sensitivity mode, case auto-correction; as variants of a string type, types "file" and "directory" are defined, which values are input in a browse dialog and formatted according to path denotation rules in the given OS (and thus are OS-dependent);
- ***enumeration types***: allow to introduce types with fixed sets of values, to input values by selection from a list of alternatives, to substitute values denotations on output.

Tools Control Languages

A utility control language is formally described by two ***layers*** of definitions:

1. ***interface***, that defines:
   - named *types* of usable parameters, with attributes:
     - *set of values (domain)*, that a parameter may accept;
     - *output representations of values* – a format to write them into a control line;
     - *input representation of values* – a way and a format to input them in a setup dialog;
   - named *formats* to convert parameters into control lines options;
2. list of ***contexts***, that define:
   - all usable parameters of this context, with attributes:
     - parameter names:
       - internal *identifier (id),* denoting a TSL `parameter`-object;
       - output *option identifier (outid)*, to be passed to a utility;
       - *explanation* to an input field *(label)* in a setup dialog.
     - parameter's type name;
     - name of a format to convert a parameter into an option;
     - parameter's *default value*;
     - parameter's extra attributes;
   - *formats* of all controllines generated by this context;
   - contents of the setup dialog of this context;
   - extra special components of this context.

Such stratification is caused by the fact that basic types of values and general formats of options in command languages of many existing programs are either identical, or very similar (compare for example, Unix-style of options, like `-option=value` or MSDOS-style, like `/option:value`). They are also expected to be the same for all utilities in one package. Therefore in all common elements of command languages which can be used by several utilities are taken out in the *interface*, while the *context* contains language elements specific to each tool.

Interfaces

An `interface` object contains definitions of parameter types and of formats to convert parameter names and values into control lines options.

A ***mechanism of definitions inheritance*** is provided for interfaces. Each interface is always ***derivative*** from some other (unique) ***basic*** interface. All definitions from the basic interface (including those inherited by itself) ***penetrate*** into (or, are ***inherited*** by) a derivative interface, adiitionally to its own definitions.

Inheritance helps to save definitions by taking their common elements out into basic interfaces and inheriting them by derivative interfaces.

By default, the basic interface is a predefined `BasicInterface` (supplied in the file `BasicInterface.xml`), where several commonly used basic types and formatting styles are defined. So, all interfaces are indirectly derivative from `BasicInterface` and may use its definitions. `BasicInterface` is the only interface which is not derivative.

Contexts

There are 4 kinds of TSL `context`-objects: `installation`, `package`, `project` and `tool`, that differ only in their purpose and some specific attributes.

A context definition consists of the following sections:

- parameters sections, containing definitions of parameters avaialable in this context;
- `input` section, containing definition of the context's setup dialog;
- `output` section, containing specification of control lines generated by the context.

Two kinds of ***activation*** are provided for a context with the following actions:

- setup dialog activation – for all parameters listed by groups in the `input` section input representations are built; the setup dialog is displayed where input fields may be modified accoding to their input kinds; input values are read and checked against parameters' virtual types; input data are saved in the Eclipse metadata storage;
- control lines generators activation – all control lines listed in the `output` section are generated according to their formats and are written into output files.

The first kind of activation is performed on calling setup dialogs in the design menu; the second kind – on exit from the setup dialog by **Ok** button, and for a `tool`-context – also on running a tool by the **Run** button.

For a `tool`-context, a ***mechanism of inheritance*** is defined. Its purpose is to simplify definitions of multiple tools calling the same utility with only slightly different settings by getting rid of huge common parts duplication. The mechanism allows to specify some `tool`-context as basic for some other context's stucture and then only remove, add, or modify some definition elements.

Design Menu (Tools Menu)

In the ***design (tools) menu*** the available tools are listed, which are organized in a tree-like hierarchy according to proper design technique.

A TSL `menu`-object consists of ***menu items*** `menuitem` (corresponding to tools) and other `menu`-objects (corresponding to ***submenus***).

A ***mechanism of edition*** is provided for menus. A menu may be set as a base for other derivative menu or submenu construction; then some items may be removed, modified or added.

Patterns

The ***mechanism of patterns expansion*** is the main and the most powerful TSL work mechanism, that allows to flexibly build string values using other string values. Its main application is construction of output representations of options and control lines; besdides, it is also used for building list values and conditional values; and it can be also used to compose conditional explanations for input fields.

A ***terminal string*** is a text containing no patterms.

A ***format string*** is a text that may contain patterns.

A ***pattern*** is a special substring of a format string, that may be replaced (or ***substituted***) by some other text, called *pattern value*.

Expansion of a format string is a process of format string transformation to a terminal state, which consists in subsequent substitution of patterns by their values. Expansion may be recursive if patterns values contain other patterns.

There are the folllowing kinds of patterns in TSL:

- ***patterns-parameters*** are TSL parameters defined in any context (including `installation`); the values of these patterns are just values of these parameters;
- ***patterns-generators*** are «pseudo-parameters» predefined in Eclipse/ExDT, which values cannot be set by user, but come from the environment (see the list in *«5. TSL specification»*);
- ***pattern-repetitor***, that serves to build list values;

- ***pattrens-options*** are TSL parameters used in control lines formats; their values are expanded format strings for options.

Conditional Clauses

There are two kinds of ***conditional clauses*** (or ***conditionals***) in TSL.

Structural conditionals are similar to conditional compilation directives in programming languages. They denote that definitions enclosed within their bounds must be processed only if given conditions are true.

Conditional expressions are similar to patterns in that sense, that they are expanded to some resulting string. In a conditional expression, several possible values are listed together with mutully exclusive conditons indicating which of these values gives the result of the whole expression.

All condional clauses are interpreted dynamically, i.e. at the moment of value extraction or at the moment of processing the definition containing this conditional. This allows to flexibly vary the structure of definitions: e.g., hide those parameters in a setup dialog, which have no sense with the tool operation mode defined by some other parameter, and also not to pass respective options in a command line.

TSL Specification

Notation

The following metalinguistical notation (EBNF) is used for TSL syntax definition, with metasymbols and metadefinitions outlined by font and color:

defined-concept → definition	– a concept definition rule
[fragment **]**	– single optional fragment
{ fragment **}**	– possibly repetitive fragment
fragment **|** … **|** fragment	– selection of alternative fragments

If a concept is defined in another section, the number of this section is indicated after its name in parentheses.

The final text derivative by the TSL grammar should conform to the general syntax of XML. The most important rules are:

- names of tags and attributes are case-sensitive;
- each tag-opening bracket `<tag attributes>` must be closed by matching `</tag>`
- reduced notation: `<tag attributes></tag>` ⇔ `<tag attributes />` ;
- double quotes, apostrophes and < sign in strings: `&qt; , &ap; , &lt;`
- attribute names in tags must not duplicate.

1. General TSL Specification Structure

The complete ***TSL specification*** of the Eclipse/ExDT configuration is composed by joining definitions from separate TSL files. These are all files locating in the `tools` subdirectory of the plug-in's installation directory, which have extension **.xml**.

Each TSL file consists of definitions of one or more top-level TSL objects: `interface`, `menu` and 4 kinds of ***contexts*** – `installation`, `package`, `project-template` and `tool`, – which contain nested definitions of other lower-level objects.

TSL-file →

```
<?xml version="1.0" encoding="UTF-8"?>
<vdt-project>
   { interface(2) | menu(3) | context() }
</vbt-project>
```

The objects arrangement by files and the order of their descriptions within a file are not important, also for objects referring one to another. However, definitions of top-level objects in each files must be complete, and in the united specification there must be no multiple definitions of the same object (the object is identified by its tag and attribute `name`).

2. Interfaces

Definition of ***interface*** specifies the abstract part of the utility control language: ***data types*** describing parameters values and и ***options formats*** in the utility's command lines.

interface →

```
<interface name="interface-name"
          [ extends="basic-interface-name" ] >
  { type(2.1) | format(2.2) }
</interface>
```

Mandatory attribute `name` specifies the unique interface name used for references from **context(4)** or from a derivative interface.

Optional attribute `extends` specifies basic interface name used for reference to another interface, inherited by this one. If this name is omitted, the interface inherits `BasicInterface`. Interface inheritance is equivalent to including the contents of the basic interface into the definition of the derivative interface. New internal objects (types and formats) must not duplicate inherited ones by names (for interfaces there is no mechanism of definitions overriding provided for other objects!).

2.1. Data Types

Definition of a ***data type*** introduces a set of allowed values, a value input method and a format to convert a value from input into internal and output representations.

type-definition →

```
<typedef name="type-name"
        [ list=("true"|"false") ] >
  type-structure
</typedef>
```

type-structure →

```
      numeric-type(2.1.1)          |
      boolean-type(2.1.2)          |
      enumeration-type(2.1.3)      |
      string-type(2.1.4)
```

Mandatory attribute `name` specifies the internal type name used for reference from **parameter(4.2)** and **enumeration-type(2.1.3)**. All used types must be defined in the interface section as named objects.

When optional attribute `list="true"`, the type is defined as a list of elements which type is described by **type-structure**. Parameters of a list type have a compound value. Input of a list parameter is performed in a separate list input dialog. To generate an output representation of a list, a format string must contain a **pattern-repetitor(2.2.1)**.

2.1.1. Numeric Types

A ***numeric type*** defines a range of integers or fixed-point real numbers.

numeric-type →

```
<paramtype kind="number"
           lo="integer"
           hi="integer"
```

```
            format="format-string"
/>
```

All attributes are mandatory. The value of low bound `lo` must be less than of high bound `hi`.

The internal representation of a numeric value is a string containing the decimal spelling of the number. Attributes `lo` and `hi`, and also all values used in utilities description are specified in TSL in their internal representation.

The format string is used to convert values of this type from input representation into internal and from internal into output represntation. Using a format line, it is thus possible to emulate fixed-point real numbers: a point in the format string separates whole positions from fractional on output only, internally real values are stored as scaled integers – Not implemented yet. The format line will be defined later; right now it is ignored and a number is output as a string of its significant decimal digits.

In the setup dialog, a numeric parameter is input in an edit control according to its format. The boundaries check is performed on input.

2.1.2. Boolean Types

A ***booleanc type*** has common semantics. It is possible to define several boolean types differing by output representations of their values.

Boolean-type →

```
<paramtype kind="bool"
           formatTrue="string"
           formatFalse="string"
/>
```

All attributes are mandatory.

In the setup dialog, a boolean parameter is displayed and input as a check box.

The internal representation of two values are strings `"true"` and `"false"`.

The format lines specify output representations of two values separately.

2.1.3. Enumeration Types

An ***enumeration type*** defines a set of fixed values. These may be selected values of any other type.

enumeration-type →

```
<paramtype kind="enum"
           base="base-type-name">
  { enumeration-value }
 </paramtype>
```

enumeration-value →

```
<item value="enumeration-value"
     [ label="explanation" ]
/>
```

Mandatory attribute `base` specifies the base type's internal name, that is the type of values used in enumeration. If values are specified by their text appearance, it is recommended to use a string type as a base.

Mandatory attribute `value` specifies the enumeration value written in the internal representation as defined for the base type. Conversion to the output representaton is also performed by rules for the base type. Optional attribute `label` specifies a string representing the value on input; if it is omitted, the input representation is build by rules for the base type – usually will be as written in `value`.

In the setup dialog, an enumeration parameter is input with a drop-down selection list containing all elements of the enumeration.

2.1.4. String Types

A ***string type*** defines a set of arbitrary text values.

string-type →

```
<paramtype kind="string"
          [maxlength="integer"]
          [textkind= ("text"|"file"|"dir")]
          [sensitivity=("insensitive"|"sensitive"|"uppercase"|"lowercase")]
          [filemask="string"]
 />
```

Only attribute `kind` is mandatory.

Attribute `maxlength` specifies the maximal string length (in fact, it is only the limitation of the input field length). By default – 256.

Attribute `textkind` indicates special flavours of a string type having different input methods. A usual text (by default, `"text"`) is input in an edit control. A file name (`"file"`) and directory name (`"dir"`) are input using a "browse" button; their input representation is built using rules to write file names and paths in the given OS.

Attribute `filemask` has sense only for files and specifies a file mask for browse dialog.

Attribute `sensitivity` specifies string case conversion. With `"sensitive"` and `"insensitive"` the input line is not converted; these two values set two case sensitivity modes for comparison of the string with the default value. With `"uppercase"` and `"lowercase"` the string is converted on input to uppercase or lowercase; the same is done with default values specified for parameters. Strings comparison is then performed in the same case mode. The default mode is `"insensitive"`.

2.2. Options Formats

An ***option format*** defines how a single parameter and its value are written to a control line of a utility.

option →

```
<syntax name="option-name"
        format="format(2.2.1)"
/>
```

All attributes are mandatory.

The internal option name is used for references to the format from **parameter(4.2)**.

The format string specifies how an output representation of a parameter is constructed. To substitute parameter's name and value into an option, use patterns-generators `%%ParamName` and `%%ParamValue`.

2.2.1. Format Expansion

format →

{ *fixed-text* **| pattern }**

pattern →

pattern-generator |
patter-repetitor

pattern-generator →

`%%`*generator-name*

pattern-repetitor →

```
%( text-repetitor %| text-separator %)
```

To expand a **format**, its string is taken as a base; then all patterns inside are expanded by the following rules:

1) A pattern-generator expands to a single string or to a list of strings. A single string is substituted as a pattern value. A list may be only substituted into a pattern-repetitor. All of predefined generators are listed in 5.

2) A pattern-repetitor consists of two parts: text-repetitor and text-separator. Inside the text-repetitor, one and only one pattern-generator is mandatory, which supplies a list of strings. Each line from this list is substituted instead of the pattern-generator into the text-repetitor; the resulting strings are concatenated together separated(!) by the text-separator. The final string replaces the whole pattern-repetitor in the format.

Sample: the pattern "%(&qt;%%SourceList&qt;%|;%)" generates a list of project files, each name quoted and separated by a semicolon.

3. Design Menu

A ***menu*** specifies the appearance and functionality of the design menu.

menu →

```
<menu name="menu-name"
      label="menu-title"
     [tip="tip-help"]
     [icon="icon-file"]
     [inherits="base-menu-name"]
     [visible=("true"|"false")]
     [after="item-name"] >
  { menu | tool }
</menu>
```

tool →

```
<menuitem name="menu-item-name"
          label="tool-title"
         [icon="icon-file"]
         [visible=("true"|"false")]
         [after="tool-name"] >
          call="tool-context-name" />
```

Nested tools and menus define the order of items and submenus in the menu.

Mandatory attribute `name` specifies the unique menu name or menu item name name used for refernces to the menu and its elements when editing.

Mandatory attribute `label` specifies the explanatory text for a submenu node or for a tool node in the menu image.

Optional attribute `tip` has sense only for the top-level menu; it specified the text of a tip, or a context help, displayed when a cursor points to the menu title.

Optional attribute `icon` specifies the path and the file name of an icon used to mark the menu item; by default, standard icons for a tool and a submenu are used.

Attribute `call`, mandatory for a tool, specifies the name of the `tool`-context which defines the parameters of the tool.

Optional attribute `inherits` indicates that this (sub)menu is created as derivative from the basic menu which name is given by this attribute. The basic menu must be a top-level menu, that is, not nested one.

The mechanism of edition, that helps creating a derivative menu from the basic menu, works as follows:

- The whole structure of the basic menu is copied in the initially empty derivative menu, and the root of this menu get the name of the derivative menu.
- Then the copied tree (called basic) is compared with the specified structure of the derivative menu; thus two nodes are considered equal if their names are the same and their parent nodes are equal (in the same sense).
- For each pair of equal nodes, *edition* is performed by the following rules:
  - *rule for attributes replacement* in a node:
    - values of attributes in a basic node are replaced with values of the same attrbutes in the derivative mode;
    - attributes, absent in the derivative node, preserved their basic value;
    - attributes, absent in the basic node, are added from the derivative;
  - *rule to delete and restore a deleted node*: is implemented via the mechanism of visibility (attribute `visible`) by the replacement rule for this attribute;
  - *rule to add a node*: if the derivative node has descendances (items and submenus), not equal to basic ones, they are added to the derivative node in the order of definition; the item is inserted:
    - after the item indicated by attribute `after`;
    - as the first item, if `after="first"`;
    - as the last element, if `after` is absent.

In particular, if there are no attributes other than `name` and `inrerits` in the definition of the derivative submenu, then the base menu is just attached as a submenu with the given name to its parent menu.

4. Contexts

Definition of a ***context*** concretizes the description of the utility control language by specifications of ***parameters***, ***setup dialogs*** and ***control lines*** at some level of operation.

context →

```
< ( installation | package | project | tool )
      name="context-name"
      interface="interface-name"
      special-attributes(4.1)
>
  parameters-section(4.1)
  input-section(4.2)
  output-section(4.3)
  special-sections(4.1)
</( installation | package | project | tool )>
```

Context tags match to its kind.

Internal section of the context may be defined in any order.

Mandatory attribute `name` specifies the context name used for references from **tool** and also for composing setup lists in the design menu of Eclipse/ExDT.

Mandatory attribute `interface` specifies the name of the interface, where types and options formats for parameters of this context are specified.

Each context, additionally, has its **special-attributes(4.1)** and **special-sections (4.1)**.

4.1. Special Attributes And Sections Of Contexts

4.1.1. Special Attributes And Sections Of Context `tool`

special-attributes →

```
( exe | shell )="utility-name"
[ package="context-package-name" ]
[ project="context-project-name" ]
[ inherits="context-tool-name" ]
[ ignore="regular-expression" ]
[ log-dir="parameter_name_of_type_string_kind_dir" ]
[ result="parameter_name_of_type_string_kind_text" ]
[ state-dir="parameter_name_of_type_string_kind_dir" ]
[ restore="parameter_name_of_type_string" ]
[ save="parameter_name_of_type_string" ]
[ autosave="parameter_name_of_type_boolean" ]
[ disable="parameter_name_of_type_boolean" ]
[ priority="floating_point_number" ]
[ abstract="true|false" ]
```

Mandatory attribute `exe` specifies the utility name – name of the program's executable file, with extension, without path. If "shell" is used instead of "exe" then utility is considered to be a shell program and the all the generated command line parameters but the first one will be merged and passed to the shell program. The first parameters is left for the shell parameters. Using shell program allows to use a pipe of multiple commands, like using grep to filter the utility output.

Optional attribute `package` specifies the name of the `package`-context, which settings are used by the utility (following visibility rules).

Optional attribute `project` specifies the name of the `project-template`-context name, which settings are used by the utility (following visibility rules).

A utility may also use settings from the installation-context without special notes.

Optional attribute `inherits` specifies the name of the `tool`-context, which settings are inherited by this derivative context. The inheritance rules are described in 4.5. This attribute is incompatible with attributes `package` and `project`.

Optional attribute `ignore` provides a regular expression pattern to ignore files by the `%%FilteredSourceList` generator. It can be set to remove library primitives from the list, like ".*unisims.*" for Xilinx Verilog primitives library. This attribute can reference other parameters so it can be configured at run time.

Optional attribute `log-dirs` designates sub-folder (relative to the project top directory) to save tool log files. This feature allows to play back the tool log files with different external parser settings that used when the tool was actually run (that may takes hours in some designs). Combined with Eclipse native regex pattern matching this provides flexibility in dealing with large tool output logs efficiently.

Optional attribute `result` specifies a file name (relative to `state-dir` described below) to save tool states. This file (usually a compressed archive) allows to save and restore environment at different stages of running tools. When the tool state is saved the actual file gets a timestamp included in the name, and Eclipse virtual link with the specified name (`result`, no timestamp) is created to point to the latest state. It is possible to skip archiving, in that case the specified state exists (is valid) until any other state-changing tool runs. Some tools like report tools or bit-stream generation does not change the design files and can run in any order.

Optional attribute `state-dir` designates sub-folder (relative to the project top directory) to save tool state files. It is conveniet to keep parameters that define `result` and `state-dir` in the project context, because multiple tools need to have access to them.

Optional attribute `restore` specifies (through a parameter value) a tool that restores this tool state. Usually this involves copying archive file to the remote host (where tools run), unpacking it or providing to the tools if they support creating/reading back snapshots.

Optional attribute `save` specifies (through a parameter value) a tool that saves this tool state. Usually this involves creation of the snapshot (using tools functionality or just archiving files)

and copying archive file (set with `result` attribute with attached timestamp) to the local directory (set with `state-dir` attribute). When the file is copied, Eclipse linked resource is created pointing to this new state file. Link name does not have the time stamp, it is exactly as specified by the `result` attribute.

Optional attribute `autosave` points to a parameter of a boolean type (it is considered 'false' if attribute is missing) that instructs VDT to create and save a snapshot after the tool completes successfully. It is also possible to save current state manually using as “save” button (floppy disk) in the user interface.

Optional attribute `disable` points to a parameter of a boolean type (it is considered 'false' if attribute is missing). If `disable` is true, this tool is not considered for automatic launch when running multiple commands. For example, it can be used to prevent timing analysis from running after the synthesis tool finishes.

Optional attribute `priority` specifies a floating-point number (default is 1.0) to determine which of the “report” tools (tools that do not change state) to run first when the current state is reached. The lower the value, the sooner the tool will be automatically launched.

Optional attribute `abstract` is a convenience way not to launch a prototype tool (tool inherited by other ones), it can have value of “true” (missing considered to be false). For example it is possible to create a timing analysis tool prototype (having `abstract`=”true”), and then “instantiate” it as post-synthesis and post-implementation variants (these variants may have either common or individual parameter values).

special-sections →
extensions-section

extensions-section →
<extensions-list>
{ extension }
</extension-list>

extension →
<extension mask=”*extension*” />

Extensions-section specifies the list of file extensions mask, for which this context may be activated.

special-sections →
actions-section

actions-section →
<action-menu>
{ action }
</action-menu>

action →
<action [label= ”*action-label*”]
[resource= ”*action-resource*”]
[check-extension=”*false*”|”*true*”]
[check-existence=”*false*”|”*true*”]
[icon= ”*action-icon*”] />

Actions-section specifies the list variants of launching the utility, such as running simulator for the selected file or for the project main test fixture. Each action is represented on a contest menu for the selected tool and on the tool bar.

`Label` specifies the text that appears in the context menu (or as a tool-tip for the toolbar icons), if absent `label` defaults to “Run for”.|

Item `resource` specifies the resource to apply the tool to, it can use generators (such as %%SelectedFile) or be a value of a configurable parameter. The name (last segment of the file path) of the `resource` appears after the `label` in the context menu, it can be empty string for some options that do not require target file to be specified (for example – remove all simulation results).

Boolean parameter `check-extension` specifies if the `resource` should be checked to match specified extensions for the tool set up in the **extensions-section**. If the resolved value of the `resource` path does not match any of the extensions this action item is disabled in the menus.

Boolean parameter `check-existence` specifies if the `resource` should be verified to exist in the file system. If the resolved value of the `resource` path does not exist this action item is disabled in the menus.

`Icon` specifies name/path of the icon used for the selected action, by default the system icon is used.

If the whole **actions-section** is missing it defaults to a single action compatible with the previous VDT version:

```
<action-menu>
    <action label=           "Run For"
            resource=        "%%SelectedFile"
            check-extension="true"
            check-existence="false"
            icon=            "System default "run" icon (triangle)" />
```

special-sections →
depend-section

depend-section →

```
<depends-list>
  { depend-item }
  ...
</depends-list>
```

depend-item →

```
<( depends files="file-list" | depends state="state-file")/>
```

Depend-section specifies the resources needed for this tool to run. Resources may be source files (HDL files, constraints files) and tool states. VDT compares time stamps for the state files and modification stamps for the source files to determine if the state of the tool is current or "dirty" and what tools need to run to satisfy dependencies of the selected tool. It is possible to "pin" tools (individually with the tool context menu or globally with the "pin" button) so they will not be automatically launched even if the files they depend on are modified later. The pinned state is automatically activated if the tool state is restored manually using the tool context menu (you may restore the latest tool state or select from multiple files).

4.1.2. Special Attributes And Sections Of Context `project`

special-attributes →

```
    label="explanation"
  [ package="context-package-name" ]
```

Mandatory attribute `label` specifies the text of explanation, which identifies this context in the drop-down list of projects in the design menu of Eclipse/ExDT.

Optional attribute `package` specifies the name of the `package`-context name, which settings are used by the utility (following visibility rules).

special-attributes →
none

4.1.3. Special Attributes And Sections Of Context `package`

special-attributes →

```
label="explanation"
```

Mandatory attribute `label` specifies the text of explanation, which identifies this context in the drop-down list of packages in the design menu of Eclipse/ExDT.

special-sections →

none

4.1.4. Special Attributes And Sections Of Context `installation`

special-attributes →

```
label="explanation"
menu="menu-name"
```

Mandatory attribute `label` specifies the text of explanation, which identifies this context in the list of installation settings in the design menu of Eclipse/ExDT.

Mandatory attribute `menu` specifies the name of the menu, which is displayed by Eclipse/ExDT, when none of user projects is opened yet. Now it must be specified, but is not used, and the menu window is empty in this case.

special-sections →

none

4.2. Parameters Section

Specification of a ***parameter*** completely defines its input, internal and output representations.

parameters-section →

{ parameter }

parameter →

```
<parameter id="parameter-name"
           [ outid="option-name" ]
           [ label="input-field-label (short)" ]
           [ tooltip="tooltip-text (long)" ]
             type="type-name"
             format="format-name"
             default="value"
           [ readonly=("true"|"false") ]
           [ visible=("true"|"false") ]
           [ omit="value" ]
/>
```

Mandatory attribute `id` specifies the name of the parameter used for references from **input-section(4.3)** and **output-section(4.4)**. Usually it is also the output option name.

Optional attribute `outid` specifies the output option name, if it differs from the parameter name. (It is necessary in case of multi-functional options, when it is worthwhile to separate it into multiple parameters differing by sense, type and/or attributes).

Optional attribute `label` specifies the explanatory text for the input field of the parameter in the setup dialog, dialogs look better if the labels are short. Longer explanations can be provides as tooltips. The `label` attribute may be omitted for invisible parameters (`visible="false"`).

Optional attribute `tooltip` specifies the tooltip text for the input field of the parameter in the setup dialog.

Mandatory attribute `type` specifies the type name used for reference to parameter's **type-definition (2.1)**, which sets its domain and formats.

Mandatory attribute `default` specifies the parameter's default value in internal representation, compatible with its type. This attribute can be used to pass values from expressions or other parameters. It may be needed for example if the same parameter value must appear in the output in different formats.

Additionally `default` may be used to enter the contents of a file or result of executing an external command to the parameter value. If the resolved value of the `default` attribute starts with a single "@" then the rest is considered to be a file name (relative to the project or absolute). The file is read and the contents (or a first line of it if the "list" attribute of the parameter type is "false") is assigned to the parameter. This value can be used in other `default` assignments, some of the attribute values, output sections and conditional expressions.

If resolved value of the `default` attribute starts with "@@" then the rest is considered to be a command with optional arguments – line is split by white spaces, so it works only for simple commands.

Parameters are evaluated after multiple events, not just once the command is launched, so long command execution will slow down the overall program performance.

The "@" may be escaped by "\" if they are needed in the first two positions of the value string. This escaped "\@" is only applicable to the first two positions of the result string.

Optional attribute `readonly="true"` disables value input in the dialog. The input field is thus visible (in gray) and shows the default or the last input value, but is not available for modification. Default value is `"false"`.

Optional attribute `visible="false"` hides forbids displaying (and thus modifying) the parameter in the dialog. Default value is `"false"`.

The attribute of visibility allows to introduce parameters with constant values, whose input is not necessary, and also parameters with conditional values, that are defined automatically depending on other parameters values.

Optional attribute `omit` specifies the value with which the parameter is not passes into an output line at all. (This is necessary in case when passing no option to a utility has other sense than passing it any definite parameter, including the default one). In this case, the output representation of the parameter is an empty string.

Joint fragments of the list of **parameters** may be enclosed into **structural conditional clauses (6)**. String values of attributes `default`, `omit`, `visible` and `readonly` may be **conditional expressions (6)**.

4.3. Input Section

The ***input section*** specifies what parameters must be configurable in the input dialog. Parameters in the section may be arbitrarily joined in semantic groups.

input-section →

```
<input [ label="dialog-title" ] >
 { input-group }
</input>
```

input-group →

```
<group [ name="group-name" ]
       [ label="tab-title" ]
       [ visible=("true" | "false") ]
       [ weight="floating-point-number" ] >

 { input-element | insert-section | delete-section }
</group>
```

input-element →

```
("parameter name" | "---")
```

insert-section →

```
<insert after="parameter-name" ] >
 { input-element }
</insert>
```

delete-section →

```
<delete>
 { input-element }
</delete>
```

Optional attribute `label` of the **input-section** specifies the title of the input dialog. Default value is "*context* Preferences", where *context* is a context kind.

Parameters, united in one group, are displayed in one tab of the input dialog, in the order of their definition. The order of tabs in the dialog is the order of groups definitions in the input section.

Group's attribute `label` specifies the tab's title; it is mandatory for all contexts, except `tool`.

Group's attribute `name` is defined only for a context; it specifies the internal name of a group used in a derivative context to edit lists of parameters and groups (see 4.5). It may be omitted if a group is not supposed to extend. If values of `name` and `label` are the same, `label` may be omitted.

Group's attribute `visible` specifies the tab's visibility mode and thus availability of this group for setup. In a `tool` context, this attribute is used to define a derivative context (4.5). In other contexts, visibility of a group may be controlled by a conditional value.

Group's attribute `weight` allows to control the order the sections appear in the final menu. It is convenient when you want to put new section before the sections inherited from the base tool. Default weight is 1.0, the higher the weight, the later this group's tab will appear in the menu.

Input-element can be either parameter name or a separator (will appear as horizontal line in the dialog). Separator is specified by a fixed text - a sequence of 3 dashes (`"---"`).

Joint fragments of the list of **input-groups** and **input-elements** may be enclosed into **structural conditional clauses (6)**. String values of attribute `visible` may be a **conditional expression (6)**.

Semantics of **insert-section** and **delete-section** is described in 4.5.

4.4. Output Section

The ***output section*** specifies what control lines are passed to the utility from this context and which parameters participate in building options for each of these lines. In the current implementation when shell program is used, the first **control-format** line is used for the shell parameters (like "-c" in "/bin/bash -c <command>"), all other **control-format** lines are merged and passed to the shell program. There may be 4 types of ***control-line*** in the ***output-section***:

- *external program* with parameters (usually a shell script) – these ***control-lines*** do not have ***dest*** attribute.
- *command files* with the ***dest*** attribute pointing to the separately defined ***parameter*** with the type of *string* kind and *file* text-kind – the parsed body of the line will be written to the specified file.
- *terminal script* – a sequence of commands to be sent to an already openexternal program session. This type on ***control-line*** has ***dest*** attribute pointing to a ***parameter*** with the type of *string* kind and *text* text-kind. The text value will be used to find an open console with the name starting with this string, and the assembled body of the blocks will be sent to an external program attached to that console.
- *parser* – an external program (similarly to described above) to pre-process terminal session output to be used by Eclipse errors/warnings/info pattern-matching. Parser can be

any program reading data from *stdin* – it can be a simple *grep* command or a custom Python script. Parser ***control-line*** does not have ***dest*** attribute (like *external program*), and it is recognized by being referenced by one of the *terminal script* blocks in their ***stdout*** or ***stderr*** attributes.

Command files are created before any other ***control-line*** blocks. External programs and terminal scripts are executed in the order of appearance sequentially, parsers are spawned as parallel processes in separate Eclipse process consoles when executing terminal scripts that use them.
All executable blocks but the very last one (external programs and terminal scripts) wait for completion of the preceeding one.

output-section →

```
<output>
 { control-line }
</output>
```

control-line →

```
<line [ name="line-name" ]
      [ dest="destination-parameter-name" ]
      [ sep="options-separator-string" ]
      [ mark="mark-string" ]
      [ prompt="finish-string" ]
      [ timeout="timeout-in-sec" ]
      [ stderr="name-of-error-parser-control-line" ]
      [ stdout="name-of-output-parser-control-line" ]
      [ errors="eclipse-errors-pattern-string" ]
      [ warnings="eclipse-warnings-pattern-string" ]
      [ info="eclipse-info-pattern-string" ]
      [ instance-capture="eclipse-info-pattern-string" ]
      [ instance-separator="eclipse-info-pattern-string" ]
      [ instance-suffix="eclipse-info-pattern-string" ]
      [ log="log_file_suffix" ] >
  { control-format | insert-section | delete-section }
</line>
```

insert-section →

```
<insert after=control-format ] >
 { control-format }
</insert>
```

delete-section →

```
<delete>
 { control-format }
</delete>
```

Attribute `name` is defined and is mandatory only for a `tool` context. It specifies the internal name of the line used in a derivative context to edit lists of options and lines (see 4.5). Terminal scripts use name of the parser they use.

Optional attribute `dest` specifies the name of the destination parameter of a file string type (`kind="string" textkind="file"`), whose value is the name of the command file, where the control line will be written. For terminal scripts this parameter is a string type (`kind="string" textkind="text"`) specifying the name (beginning of the full name) of the console where the referenced program is opened.

If `dest` attribute is omitted then the destination is the utility's command line (one of) or the external parser name.

The control line is composed by concatenation of strings resulting from control-formats(4.4.1). If attribute `sep` is present, the separator string is inserted between each two adjacent strings.

Control format strings are included in the body of the XML text node and are parsed by the plug-in that counts opening and closing double quote marks (″) and recognizes escape by the backslash, so for example `″\″″` will result in a single-character (″) control format string. Other recognized control characters in control-formats are \n, \t and \xNN where NN is a hexadecimal character code. It is different for the attribute values – they are processed by the XML parser, so it is possible to enclose strings in single-quotes or to use " when the double quote (″) is needed as part of the value. It is not possible to use `\″` there because that would result in invalid XML code.

Optional attribute `mark` specifies the character sequence that will be removed just before outputting the assembled result to a file or passed to a program. This allows to protect empty lines or leading/trailing spaces (as needed for Python programs, for example) from being trimmed during TSL elaboration.

Optional attribute `prompt` (applicable to *terminal script*s) specifies the string that can be used to determine that the script has finished. The script body has to make sure the program will generate it. For example in TCL session the last line of the script can be "puts '@@FINISHED@@'" and `prompt="@@FINISHED@@"`.

Optional attribute `timeout` (applicable to *terminal script*s) specifies how long to wait for the terminal script to finish (in integer seconds), the script will be considered finished after whatever comes first – prompt is detected in combined stdin+stdout stream or timeout is reached. After that the optional attached parser processes will be terminated and the execution will proceed to the next ***control-line*** block. No actions will be performed to the terminal program itself – it may continue if not done already.

If none of the `prompt` or `timeout` are specified the default `timeout="1"` is applied.

The `stderr` and `stdout` specify the names of external program parsers used to process the terminal script results by temporarily connecting to the stderr and stdout streams of the process running in the console (specified by the The `stderr` and `stdout` specify the names of external program parsers used to process the terminal script results by temporarily connecting to the stderr and stdout streams of the process running in the console (specified by the `name` attribute). If only `stdout` is provided then both stdout and stderr will be copied to the stdin of the same parser window, if only `stderr` is specified then stdout of the terminal program will go to the parser. If both are specified – two parsers will be spawned (they can be the two separate instances of the same parser ***control-line*** block) in two separate Eclipse consoles.

Attributes `errors` , `warnings` and `info` specify the names of parameters that contain patterns (regular expressions) to extract resource name, line number and the message text from the tool output. It is possible to put `SuppressWarnings` keyword in the source file (according to the language comments syntax) as a previous line before the reported one, together with `"all"` or the name of the tool it applies to.

Attributes `instance-capture` , `instance-separator` and `instance-suffix` specify names of parameters that hold regular expressions for parsing the database object references in the tool output when source code location (filename and line number) is not explicitly available in the tool output. The first one, `instance-capture` is a regular expression to detect a reference to a database object, current value for Xilinx Vivado synthesis is:

```
((([#a-zA-Z_$]([a-zA-Z_$0-9]|\[[0-9:]+\])*)(\.|:))+([a-zA-Z_$]
([a-zA-Z_$0-9]|\[[0-9:]+\])*))
```

Outer "()" specify what should remain in the problem text when object reference is removed – it is possible for external pre-processor (i.e. python script) to insert extra prefix and/or suffix to facilitate parsing of the line by VDT, these elements should be inside outer "()". The next pair (group1) encloses the object reference itself that should be parsed by VDT (the "#" in front of the

reference tells that the rest reference starts from the instance element somewhere in the hierarchy under the top module, without it VDT looks for the top level element (Verilog module).

Next `instance-separator` is just a regex to specify hierarchy separator, `"\."` by default.

Pattern `instance-suffix` matches some known suffixes that tool may append to the name specified in the source code. When trying to match the name segment VDT first removes any bit specifications, then tries to match the name exactly. If that fails it looks for database objects that match the beginning of the specified name and the remaining part of the provided name matches the specified suffix. Only if that fails also, VDT matches objects that match the beginning of the name segment it parses.

Current value of `instance-suffix` is `"_reg|__parameterized[0-9]*"`, it is likely to include more known patterns in the future.

Optional attribute `log` specifies suffix of the tool log for this output block that will be appended to the tool name with "_" separator, if this line is the only that needs logs then `log=""` is adequate (block without the attribute does not generate log file. Each stored log file includes the unique time stamp of the run, the link to the latest log is also generated, so each tool context menu offers to play back latest log or the use the file selector dialog to select the desired one.

When log play-back is activated, program uses log files to feed them to the same external parsers (and native regex matching capability of Eclipse) defined for this tool/line, so it is possible to get output with the different filter settings than ones used during the actual tool run.

Joint fragments of the list of **control-formats** may be enclosed into **structural conditional clauses (6)**.

4.4.1. Formatting Of A Control Line

control-format →
" **{** *fixed-text* **| pattern }** "

pattern →
pattern-option |
pattern-generator |
pattern-repetitor

pattern-option →
`%`*parameter-name*

pattern-generator →
`%%`*generator-name*

pattern-repetitor →
`%(` *text-repetitor* `%|` *text-separator* `%)`

The order of formatting of a control line is similar to that described in 2.2.1. However, a pattern-parameter here works as a pattern-option, that expands not in a parameter's value, but in an output representation of the option corresponding to this parameter. Using patterns-generators relevant to parameters properties (see 6) is not allowed here.

4.5. Specification of Derivative Context `tool`

The mechanism of edition, that helps to create a derivative context from the base context indicated by attribute `inherits`, works as follows:

- For header attributes of the derivative **context(4)**:
  - attribute `name` must specify a unique name;
  - attribute `interface` must refer either to the interface of the base contex, or to an interface derived from it;

- attribute `exe` may be either omitted (then it is inherited), or contain the same utility name, or some other name. (The last may have sense, if the derived context refers to other version of the same utility installed in another directory).

- In the **parameters-section(4.2)** of the derivative context it is possible:
  - to define a new parameter with a unique identifier;
  - to modify any attributes of an existing parameter by their redefinition;
- In the **input-section(4.3)** of the derivative context it is possible:
  - to redefine the dialog's title – section's attribute `label`;
  - to define a new group with a unique name and arbitrary contents;
  - to alter visibility of an existing group (exclude from input or to include again) – group's attribute `visible`.
  - to add new **input-elements** to a group.

  The mechanism of editions for an input section works as follows:
  - First, all groups (elements) from the base context are included into the section (group) in the order of their definition; the contents of an already existing group is filled with base elements similarly.
  - Then all new groups (elements) are included into the section (group) in the order of their definition in the derivative context.
  - Then, if an **insert-section** is present in the group, all elements from it are included into this group immediately after the element indicated by attribute `after`. If `after="first"`, elements are inserted in the beginning of the group.
  - Then, if a **delete-section** is present in the group, all elements from it are removed from this group.
- In the **output-section(4.4)** of the derivative context it is possible:
  - to define a new control line with a unique name;
  - to delete a line by its redefinition with `dest=""`;
  - to add new **control formats(4.4)** in a line.

  The mechanism of edition for the output section works similar to that of input section.
- In the **extensions-section(4.1.1)** of the derivative context it is possible:
  - nothing right now.

5. Patterns-generators

Patterns-generators are "pseudo-parameters" predefined in TSL that do not need definitions or input. Their values come from the Eclipse/ExDT environment.

In strings, patterns-generators are marked with double percent signs to distinguish them from parameters.

The following generators are available in Eclipse/ExDT version 1.1.0 (colored are generators of parameters, which are not available in formats of control lines):

- Single value:

`ParamName` – this parameter's name;
`ParamValue` – this parameter's value in output representation;
`ProjectName` – current project name;
`TopModule` – main module name (for which the utility is called);
`SelectedFile` – file currently selected in Eclipse file navigator;
`CurrentFile` – current file name (for which the utility is called);
`CurrentFileBase` – current file name without the extension;
`BuildStamp` – unique number based on the tool start time;
`CurrentFile` – current file name (for which the utility is called);
`ChosenActionIndex` – index of the chosen action variant (starting from 0);
`StateFile` – name of the state file (snapshot archive);
`StateBase` – name of the state file without extension;

`StateDir` – path to a directory (relative to project root) where state file(s) are saved;
`ToolName` – name of the current tool ;
`ParsersPath` – path to the directory for parser executables/scripts;
`OS` – OS name ("Windows", "Linux", etc.);
`UserName` – current OS user name ;

- List values:

`ParamValue` – list-value parameter in output representation;
`SourceList` – files list, containing the closure of the main module's imports;
`FilteredSourceList` – files list, containing the closure of the main module's imports with `ignore` – filter (specified for the tool) applied;
`FileList` – list of all project files;
`TopModules` – files top (not unstanciated by other) modules in the `CurrentFile`;

6. Conditional Clauses

The syntax and usage of structural conditional clauses and conditional expressions are described in a separate document *"Conditional Clauses"*.