Code Style

There Must Be No Labels or Goto Statements

- **Symbolic Name in XML Configuration**: checkNoLabelsOrGotoStatements
- **Dependant Tags in XML Configuration**: -

The check makes sure that there are no labels and goto statements in the test code.

There Must Be No Nested Alt Statements

- **Symbolic Name in XML Configuration**: checkNoNestedAltStatements
- **Dependant Tags in XML Configuration**: maximumAllowedNestingDepth

The check makes sure that there are no alt statements nested within other alt statements or altstep definitions beyond a given depth (specified via the **maximumAllowedNestingDepth** configuration tag). The topmost alt or the enclosing altstep definition is considered of depth 0, first level nesting is of depth 1, and so on. The default nesting depth is 0, meaning that no alt statements are allowed within other alt statements or altstep definitions.

Example:

```
module checNoNestedAltStatements {
    altstep as_0 ()
    runs on myComponent {
        //nesting within an altstep
        [] p2.receive ( msg_b ) {
        }
        [] p2.receive ( msg_c ) {
            setverdict ( pass );
            // nested alt, level 1
            alt {
                [] p2.receive ( msg_b ) {
                }
                [] p2.receive ( msg_c ) {
                    setverdict ( pass );
                }
                [] p2.receive {
                }
            }
        }
        [] p2.receive {
            // nested alt, level 1
            alt {
                [] p2.receive ( msg_b ) {
```

```
            }
            [] p2.receive ( msg_c ) {
                setverdict ( pass );
            }
            [] p2.receive {
                // nested alt, level 2
                alt {
                    [] p2.receive ( msg_b ) {
                    }
                    [] p2.receive ( msg_c ) {
                        setverdict ( pass );
                    }
                    [] p2.receive {
                    }
                }
            }
        }
    }
}
testcase t_sendMsg ()
runs on myComponent {
    //nesting within alt statement
    p1.send ( msg_a );
    alt {
        [] p2.receive ( msg_b ) {
        }
        [] p2.receive ( msg_c ) {
            setverdict ( pass );
            // nested alt, level 1
            alt {
                [] p2.receive ( msg_b ) {
                }
                [] p2.receive ( msg_c ) {
                    setverdict ( pass );
                }
                [] p2.receive {
                }
            }
        }
        [] p2.receive {
            // nested alt, level 1
            alt {
                [] p2.receive ( msg_b ) {
                }
                [] p2.receive ( msg_c ) {
                    setverdict ( pass );
                }
                [] p2.receive {
```

```
                        // nested alt level 2
                        alt {
                            [] p2.receive ( msg_b ) {
                            }
                            [] p2.receive ( msg_c ) {
                                setverdict ( pass );
                            }
                            [] p2.receive {
                            }
                        }
                    }
                }
            }
        }
    }
```

Output

The output includes the scope of the alt statement (starting line - end line) and has the following format:

```
Alt statement nesting depth (<DEPTH>) exceeds maximum allowed nesting depth
(<maximumAllowedNestingDepth>)!
```

There Must Be No Permutation Keyword

- **Symbolic Name in XML Configuration**: checkNoPermutationKeyword
- **Dependant Tags in XML Configuration**: -

The check makes sure that there are no permutation keywords in the test code.

There Must Be No AnyType Keyword

- **Symbolic Name in XML Configuration**: checkNoAnyTypeKeyword
- **Dependant Tags in XML Configuration**: -

The check makes sure that there are no anytype keywords in the test code.

There Must Be No Modified Template of a Modified Template

- **Symbolic Name in XML Configuration**: checkNoModifiedTemplateOfModifiedTemplate
- **Dependant Tags in XML Configuration**: -

The check makes sure that there are no modified templates are derived from modified templates. In the example, myTemplate3 is a modified template of degree 2 and myTemplate4 is a modified template of degree 4. Both will be reported by T3Q.

Example:

```
module checkNoModifiedTemplateOfModifiedTemplate {
    type record MyRecordType {
        integer field1 optional,
        charstring field2,
        boolean field3
    }
    template MyRecordType MyTemplate1 := {
        field1 := 123,
        field2 := "A string",
        field3 := true
    }
    template MyRecordType MyTemplate2 modifies MyTemplate1 := {
        field1 := omit,
        field2 := "A modified string"
    }
    // MyTemplate2 is already a modified template
    template MyRecordType MyTemplate3 modifies MyTemplate2 := {
        field1 := 22
    }
    // this one is even modified two times
    template MyRecordType MyTemplate4 modifies MyTemplate3 := {
        field3 := false
    }
        // NESTED TEMPLATES
        type record MyRecordType2 {
        integer field1 optional,
        charstring field2,
        boolean field3,
        MyRecordType nestedTemplate
    }
    template MyRecordType MyNestedTemplate1 := {
        field1 := 123,
        field2 := "A string",
        field3 := true,
        nestedTemplate :={field1:= 1, field2:="a", field3:=true}
    }
    template MyRecordType MyNestedTemplate2 modifies MyNestedTemplate1 := {
        field2 := "B string",
        nestedTemplate :={field1:= 2, field2:="b", field3:=true}
    }
    template MyRecordType MyNestedTemplate3 modifies MyNestedTemplate2 := {
```

```
        field2 := "C string",
        nestedTemplate :={field1:= 3, field2:="c", field3:=true}
    }
}
```

Local Definitions Must Be Declared at the Beginning of Testcases, Functions, Altsteps, and Component Definitions, in a Specified Order

- **Symbolic Name in XML Configuration**: checkLocalDefinitionsComeFirst
- **Dependant Tags in XML Configuration**: localDefinitionTypes

This is an evolved version of the check for variables only in the same context (declared before any other statements). In this version, not only variables, but also local constants and timers can be considered (if configured). Additionally, the order of occurrence of these three classes of local definitions can be specified and checked for. With the help of the help of the **localDefinitionTypes** configuration tag, which contains a list **string** tags, the contents and the order of local definitions can be specified. The default configuration contains four possible types of local definitions: **VarInstance** (variables), **ConstDef** (local constants), **TimerInstance** (local timers), and **PortInstance** (port instances, only within components), configured in this order, meaning that constants are expected to be declared after all variables and before all timers (and ports, in the context of component definitions):

```
<localDefinitionTypes>
  <string>VarInstance</string>
  <string>ConstDef</string>
  <string>TimerInstance</string>
  <string>PortInstance</string>
</localDefinitionTypes>
```

This order can be changed by moving the configured entries up and down the list. If timers and ports are to be disregarded, for example, then the whole configuration tags shall be omitted, so that the configuration will be:

```
<localDefinitionTypes>
  <string>VarInstance</string>
  <string>ConstDef</string>
</localDefinitionTypes>
```

Only the local definition types provided in the default configuration can be used, and only if spelled correctly, meaning that case-sensitivity matters in this context ("constdef" hence will not be recognized).

Note that control part definitions are not analyzed. This will be subject to change, where control part definitions will be analyzed as well.

Import Statements Must Be Declared at the Beginning of Modules

- **Symbolic Name in XML Configuration**: checkImportsComeFirst
- **Dependant Tags in XML Configuration**: -

Similarly to "Variables Must Be Declared at the Beginning of Testcases, Functions and Altsteps", this check makes sure that all import statements are always at the beginning of any module.

There Must Be No Duplicated Identifiers On the Module Level

- **Symbolic Name in XML Configuration**: checkNoDuplicatedModuleDefinitionIdentifiers
- **Dependant Tags in XML Configuration**: -

This check will make sure there are no identical identifiers for definitions on the module level among the analyzed modules (regardless of type).

There Must Be No Unused Definitions On the Module Level

- **Symbolic Name in XML Configuration**: checkZeroReferencedModuleDefinitions
- **Dependant Tags in XML Configuration**: zeroReferencedModuleDefinitionsExcludedRegExp

This check will make sure there are no unused definitions on the module level among the analyzed modules, including references through imports. References within import statements however will be disregarded, meaning that if a definition is imported, but never used, it will be considered an unused definition as well. Group definitions are excluded from this check. Additionally, through the **zeroReferencedModuleDefinitionsExcludedRegExp** configuration tag, a regular expression can be specified to allow certain modules (based on the module name) to be excluded from this check. The regular expression is empty by default, meaning that all modules are considered by default.

There Must Be No Inline Templates

- **Symbolic Name in XML Configuration**: checkNoInlineTemplates
- **Dependant Tags in XML Configuration**: -

This check will make sure there are no inline templates used in the analyzed modules.

There Must Be No Over-specific Runs On Clauses

- **Symbolic Name in XML Configuration**: checkNoOverSpecificRunsOnClauses
- **Dependant Tags in XML Configuration**: recursionInCheckNoOverSpecificRunsOnClauses

This check will make sure no over-specific **runs on** clauses are used. An over-specific runs on clause is when none of the component element definitions (variables, timers, constants, and ports) of the component used in the runs on clause are referenced within the body of the function or alststep with the runs on clause. If at least one of the component element definitions is referenced within the body of the function or altstep, then no problem is reported. If the **recursionInCheckNoOverSpecificRunsOnClauses** setting is enabled (which it is by default), also the functions and altsteps referenced within the body of the current construct will be inspected. This is due to the fact that often wrapper functions are used that do not make direct use of the component element definitions themselves.

As of v1.0.3, test cases are not considered in this check, due to a change request based on the notion that test cases are required to have a runs on clause, regardless of whether they actually utilize any of the component element definitions of the MTC.

Example:

```
module checkNoOverSpecificRunsOn {
    //global const for validation
    const someType someGlobalConst := 21;
        //component in question
    type component someComponent {
        var someType someVarName := 2;
        var someType someOtherVarName := 2, someAnotherVarName := 4;
        var charstring someCharStringVar := "xyz";
        var template someType someTemplateVarName := 3;
        const someType someConstName := 1;
        const integer someIntegerConstName := 42;
        const someOtherType someConstName2 := someModuleParameterName2, someConstName3 := some
        timer someTimer;
        port somePortType somePortInstance;
    }
        type component someComponent2 extends someComponent{
        port somePortType somePortInstance2;
        }
        //good
    function someFunction ()
    runs on someComponent {
        //some references to component definitions here
        someTimer.start ( 10.0 );
    }
        //bad
    function someOverSpecificFunction ()
    runs on someComponent {
                //ay(); //this will make it valid
                //wrapperfunction(); //this will make it valid
                wrapperFunction(); //unresolvable due to a typo - proper error message provide
    //no references to component definitions here
```

```
}
    //irrelevant
function someGenericFunction () runs on someComponent{
            somePortInstance.send("x");
//no relation
}
//good
//a tricky example with a referenced function that uses the relevant fields
//wraper function
function wrapperfunction ()
runs on someComponent {
    //function that uses the component definitions
    someFunction ();
}
    //bad
testcase tcx ()
runs on someComponent {
            //ay(); //this will make it valid
}
    //good
    //tricky wrapper example
altstep ax ()
runs on someComponent {
    var integer a := someFunction();
    [] ay ()
    [] ay ()
}
    //good
altstep ay ()
runs on someComponent {
    var integer a := someFunction();
    [] someTimer.timeout {
                    wrapperfunction();
    }
}
    //tricky cyclic call sequences
    function f1() runs on someComponent{
            f2();
            f3()
    }
    function f2() runs on someComponent{
            f1()
            f4()
    }
    function f3() runs on someComponent{
            f2()
    }
    function f4() runs on someComponent{
```

```
        }
    }
```

There Must Be No Unused Imports

- **Symbolic Name in XML Configuration**: checkNoUnusedImports
- **Dependant Tags in XML Configuration**: -

This is a rather complex quality check, that seeks to identify unused imports. As a first step it checks whether the imported module exists. If it does not exist, it is either because the imported module is not a part of the analyzed input, or because the name of the imported module has been misspelled. Whatever the case, if the module cannot be resolved, a corresponding message is provided and the check for that particular import statement is finished. If the module can be resolved, depending on the context, the following situations are checked:

1. If an **all** keyword is used in a non-type-restrictive manner (i.e. not associated to a certain definition type), then it is checked whether there is at least one reference of any module definition from the imported module within the importing module. Whether there are exceptions specified in the import statement makes no difference in this case, because in such a scenario, it is no longer possible to have references pointing back to the imported definition and thus the results of this check remain correct.
2. If a non-specific type import is used (i.e. an **all** keyword is preceded by definition type keyword), then it is checked whether there is at least one reference of any module definition of the given type from the imported module within the importing module. Same conditions apply as in the non-type-restrictive use of the **all** keyword. If all groups are to be considered, then only the definitions within groups are considered, those outside groups are not considered.
3. If a specific type import is used (i.e. a definition type keyword, followed by a (list of) identifiers), several checks are performed:
    1. Check if the identifier is resolvable. If it is not it means that either the definition is not present within the imported module or the identifier is perhaps misspelled.
    2. If the identifier is resolvable, it is checked whether the definition is actually within the imported module. It may be the case that the definition has already been imported from another module (see also Documentation/T3Q/Quality-Checks/Code-Style), in which case a corresponding message is provided and the actual references for that identifier are not checked any further. It may also be the case that the definition has already been imported from another module, but also does not exist in the module from which it is attempted to be imported again (the definition has been moved perhaps?!), in which case another corresponding message is provided, and no further analysis of the actual references is performed.
    3. Finally, if the identifier is uniquely and correctly resolvable, is is checked whether there are any references to that imported definition within the importing module. If a specific group is imported, it is checked whether any of the definitions within that group are referenced in the importing module instead.

Note that currently this check is rather computationally expensive and thus **disabled by default**. Once suitable optimizations are added, it will be re-enabled by default.

There Must Be No Unused Formal Parameters

- **Symbolic Name in XML Configuration**: checkNoUnusedFormalParameters
- **Dependant Tags in XML Configuration**: -

This check makes sure there are no unused formal parameters. There are several peculiarities associated with this quality check:

- External functions are excluded from this check since their formal parameters can never be used within their TTCN-3 definitions
- In the case of modified template definitions, all the formal parameters defined in the modified template (currently matched by name only) are not checked again in the modifying template. Only the additional formal parameters defined in the modifying template are checked in such a case.
- Type parametrization is currently left out, since type parametrization is moved to a package in TTCN-3 v4.1.1.
- In the case of cyclic call sequences, a corresponding *INFORMATION* message will be generated.

There Must Be No Unused Local Definitions

- **Symbolic Name in XML Configuration**: checkNoUnusedLocalDefinitions
- **Dependant Tags in XML Configuration**: -

This check makes sure there are no unused local definitions. This covers local constants, timers, variables, ports, template variables and local template definitions. Currently component definitions, function definitions, altstep definitions and testcase definitions are analyzed, as well as module control parts.

There Must Be No Uninitialised Local Variables

- **Symbolic Name in XML Configuration**: checkNoUninitialisedVariables
- **Dependant Tags in XML Configuration**: checkNoUninitialisedVariablesExclude

This check makes sure there are no uninitialised local variables. This covers local variables declared within statement blocks or module control parts. Variables declared within components are not considered as they may or may not be initialised depending on which other behaviour constructs have been executed. Instead, the data flow analysis is performed strictly within the scope of the top-level statement block. In the following example, the state of the various variables declared within a function is described for each statement using comments directly above the statement.

```
function f_ConditionalSpec() {
    var integer v_s0;
    var integer v_s1;
    var integer v_s2 := 1;
    var integer v_s3;
    var integer v_s4;
    var integer v_s5;

    //v_s2 is initialised upon declaration -> no warning
    if (v_s2 == 1) {
        //v_s3 is not initialised -> warning
        //v_s0 initialised within the conditional after this statement
        v_s0 := v_s3 + 1;

        //v_s1 is not initialised -> warning
        //v_s2 is initialised -> no warning
        //v_s3 initialised within the conditional after this statement
        v_s3 := v_s2 + 1 + v_s1;

        //v_s3 is initialised above -> no warning
        //v_s4 initialised within the conditional after this statement
        v_s4 := v_s3 + 1;
    } else {
        //v_s2 is initialised -> no warning
        //v_s4 initialised within the else branch after this statement
        v_s4 := v_s2 + 1;
    }

    //v_s3 is only initialised within one of the possible paths -> warning
    //v_s4 is initialised within both possible paths (within all branches of the condition
    v_s5 := v_s3 + v_s4;
}
```

For variables initialised within branching behaviour, such as if-else if-else constructs, alt statements, and call statements, the data flow analysis evaluates whether each branch directly initialises the variables with absolute certainty. If the variable is initialised in only some of the branches, then a warning is raised as it cannot be ensured that a variable will be initialised before use during execution. This also applies to nested branches. For loops, it cannot be ensured that a variable will be initialised as a loop can be skipped if the loop condition evaluates to false. In such case, variables initialised within the loop are considered as such in the subsequent statements within the loop, however, for statements outside the loop, the variable is still considered to be not initialised (unless there is explicit initialisation outside the loop).

The **checkNoUninitialisedVariablesExclude** setting can be used to filter out warnings for variables of certain types. The meta-types enumerated, union, record of, record, set of, set, can be listed in order to exclude warnings for all variables of the corresponding meta-type from the output, e.g. exclude all variables of all record types.

Additionally, user defined concrete types can be listed as well, so that only warnings for variables of a specific type, e.g. a specific record type, are excluded from the output. In the following example configuration, all warnings for all variables of set and set of types, as well as for all variables of the type MyRecordType? are excluded from the output.

```
<checkNoUninitialisedVariablesExclude >
  <string>set</string>
  <string>set of</string>
  <string>MyRecordType</string>
</checkNoUninitialisedVariablesExclude >
```

There Must Be No Literals

- **Symbolic Name in XML Configuration**: checkNoLiterals
- **Dependant Tags in XML Configuration**: -

This check makes sure there are no literals used, except in module parameters, template definitions, and constant definitions. Note that while in both module-level constants and local constants literals are permitted, in the case of templates, only template definitions at the module level can contain literals. Note also that currently matching symbols, boolean values, verdicts, omit-values, enumerated values, and address-values are not considered *literals*. This may be a subject to change.