T3Q Quality Checks

In the following, we describe the checks that T3Q is currently able to do. We describe them by their natural name in the section heading and with their symbolic name that is used to configure them in the configuration profile. Also we will refer to dependent tags in the configuration profile. Please note that the provided example are **not semantically complete**. They are just used to illustrate the problems. We provide sample listings where appropriate to illustrate the checks further.

Naming Conventions

- **Symbolic Name in XML Configuration**: checkNamingConventions
- **Dependant Tags in XML Configuration**: namingConventionsConfig (the whole subsection)

This quality check analyzes the identifiers for different entities (also in different contexts) and checks whether they comply to the supplied naming schemes. The naming schemes are defined as regular expressions in the "namingConventionsConfig" subsection in the T3Q configuration file. The basis for the naming conventions as well as the default values are taken from ?http://www.ttcn-3.org/NamingConventions.htm. The default settings in the "namingConventionsConfig" subsection are listed below. The fields are self-explanatory - they are formed following the schema "{languageElement}RegExp", where the language elements are also taken from ?http://www.ttcn-3.org/NamingConventions.htm and ordered in the same way as on the web page. Blank naming rules will be ignored.

```
<namingConventionsConfig>
  <moduleRegExp>[A-Z].*</moduleRegExp>
  <groupRegExp>[a-z].*</groupRegExp>
  <dataTypeRegExp>[A-Z].*</dataTypeRegExp>
  <messageTemplateRegExp>m_[a-z].*</messageTemplateRegExp>
  <messageTemplateWithWildcardsRegExp>mw_[a-z].*</messageTemplateWithWildcardsRegExp>
  <derivedMessageTemplateRegExp>md_[a-z].*</derivedMessageTemplateRegExp>
  <derivedMessageTemplateWithWildcardsRegExp>mdw_[a-z].*</derivedMessageTemplateWithWild
  <stf160sendTemplateRegExp>cs_[a-z].*</stf160sendTemplateRegExp>
  <stf160receiveTemplateRegExp>cr_[a-z].*</stf160receiveTemplateRegExp>
  <signatureTemplateRegExp>s_[a-z].*</signatureTemplateRegExp>
  <portInstanceRegExp>[a-z].*</portInstanceRegExp>
  <componentInstanceRegExp>[a-z].*</componentInstanceRegExp>
  <constantRegExp>c_[a-z].*</constantRegExp>
  <localConstantRegExp>cl_[a-z].*</localConstantRegExp>
  <extConstantRegExp>cx_[a-z].*</extConstantRegExp>
  <functionRegExp>f_[a-z].*</functionRegExp>
  <extFunctionRegExp>fx_[a-z].*</extFunctionRegExp>
  <altstepRegExp>a_[a-z].*</altstepRegExp>
  <testcaseRegExp>TC_.*</testcaseRegExp>
  <variableRegExp>v_[a-z].*</variableRegExp>
```

```
  <componentVariableRegExp>vc_[a-z].*</componentVariableRegExp>
  <timerRegExp>t_[a-z].*</timerRegExp>
  <componentTimerRegExp>tc_[a-z].*</componentTimerRegExp>
  <moduleParameterRegExp>[A-Z][A-Z_1-9]*</moduleParameterRegExp>
  <formalParameterRegExp>p_[a-z].*</formalParameterRegExp>
  <enumeratedValueRegExp>e_[a-z].*</enumeratedValueRegExp>
</namingConventionsConfig>
```

Note that component and port type definitions fall under the generic category "data type" definitions and are therefore required to follow the same naming schema (start with an uppercase letter). This may be subject to changes in the future. Note also that variables in "for" statements are also required to follow the general naming convention for all variables. Additionally, note that only message templates that directly include wildcards or matching expressions are classified as "message templates with wildcards / matching expressions". This will be subject to change in the future, in that a deep search will be performed, resolving all referenced templates and probing them for wildcards / matching expressions.

Note also that there are additional template restriction based naming conventions checks for templates. These reflect the STF160's notions of send and receive templates. Although the names may sound misleading, for these naming conventions the actual usage context (i.e. send or receive statements) for the templates in question is **not** taken into consideration. These rules are based on template restrictions in the definition of the templates, thus are also content-based. The rules according to which templates are classified as send or receive templates are as follows:

1. Templates defined with an **omit** or **value** restriction are considered send templates and shall conform to the naming convention. If such templates have formal template parameters, then these formal template parameters shall also be defined with the same (*omit* or *value*) template restrictions. If the formal template parameters are not defined with the appropriate template restrictions for this type of template (i.e. the template definition is defined inconsistently or ambiguously), an *INFORMATION* message will be provided informing of the occurrence, and the naming convention will not be checked for such occurrences.
2. Templates defined without a restriction or with a *present* restriction are considered receive templates and shall conform to the naming convention. There is no restriction on the formal template parameters for such templates.

With careful selection of the naming conventions rules, it is possible to combine the template naming conventions for wildcards / no wildcards and STF160's send / receive templates if desired. If only STF160's send / receive distinction is desired, then the naming convention rules for wildcards / no wildcards templates shall be left blank.

Structure of Data

Alphabetic Ordering of Types withing Groups

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

This quality check analyzes type definitions within groups in the same module and throws a warning if type definitions within the same group are not alphabetically (or alpha-numerically, where numbers come before letters) ordered. The ordering is case-insensitive.

Grouping of Ports and Related Messages

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

This quality check verifies that port definitions are grouped together with all the message types or signatures referenced within the port definition. The ports and the message types / signatures related to them shall be grouped within the same group in the same module. Limited nested grouping is allowed, where the message / signatures may be grouped in a subgroup within the group in which the port to which they are related was defined. This can be best illustrated by examples:

```
module checkPortMessageGrouping {
    import from checkPortMessageGroupingExternal all;
    //correct examples

    group correct {
        type port port1 message {
            in m1;
            out m2, m3;
        }
        type port port2 message {
            inout m1, m3;
        }
        type port port3 message {
            inout m4, m3;
            out m1, m2;
            in integer, charstring;
            out integer;
        }
        type port port4 procedure {
            in sm1, sm2;
            out sm3, sm4;
```

```
        inout sm5, sm6;
    }
    type port port5 mixed {
        in sm1, m1;
        out integer;
    }
    group nested {
        group signatures {
            signature sm1();
            signature sm2();
            signature sm3();
            signature sm4();
            signature sm5();
            signature sm6();
        }

    }

    type integer m1;

    type record m2 {

    }

    type set m3 {

    }

    type record m4 {

    }

}
//incorrect examples

group incorrect {
    type port port6 message {
        in m1, m2;
        out m3;
    }
    type port port7 message {
        inout checkPortMessageGroupingExternal.m5;
        out m6;
    }
    type port port8 procedure {
        inout sm1, sm2;
        out sm3;
    }
```

```
    }

    type record m6 {

    }

    type port port9 message {
        inout m6, m7;
    }

    group g1 {
        type set m7 {

        }
        type record m8 {

        }
    }

}
```

In the above module definition, the contents of group `correct` are OK, where as the contents of group `incorrect` are not, because the message types are not defined under the same group where the port they are related to is defined.

If a port is defined outside a group, it automatically means that the related message types cannot be defined within the same group and they are not further analyzed. If a message type is defined outside a group, it also means that it is not in the same group as the port definition it is related to. If a message type is defined within a group with the same name as the one where the port it is related to is defined, but within a different module, it is also considered a violation of the constraint and a warning will be thrown. If a message type related to a port type definition cannot be resolved, it is considered a violation of the constraint as well and an appropriate warning is thrown.

The standard (first) line numbers in the output indicate the reference point - on which lines within the port type definition has the violating message type been referenced. Additionally, next to the message type / signature name and the port type name, location triples <startLine,moduleName,groupName> are provided to facilitate the easier identification and localization of the elements violating the constraint.

Additionally, if nesting is present as in the above example, but does not violate the constraints, an information message will be provided in the output to inform the user of the occurrence.

No All Keyword in Port Type Definitions

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

The check makes sure that there are no all keywords in port type definitions.

Log Statements

Log Format Must Match the Format of a Regular Expression

- **Symbolic Name in XML Configuration**: checkLogStatementFormat
- **Dependant Tags in XML Configuration**: logFormatRegExp, checkLogItemFormat, processSubsequentLogStatementsAsOne

The check inspects the string in each log statement and matches it against a given regular expression. The predefined regular expression corresponds to the current ETSI guidelines. The regular expression can be adapted and customized for each profile however by changing the content of the **logFormatRegExp** tag in a profile of the XML configuration. In addition, the first regexp group is matched against the name of outer compound of the log statement, i.e., the function, testcase, or altstep name. If the first group is empty, the latter additional check will be ignored. That means, if T3Q shall not check for the containing construct, there must be an artificial empty first group in the specified regexp.

The current default regular expression is:

```
[\*]{3}\s([fta]_[a-zA-Z0-9]+?):\s(INFO|WARNING|ERROR|PASS|FAIL|INCONC|TIMEOUT):\s.*?[\*]{3}
```

The **checkLogStatementFormat** setting enables checking the *whole* log statement where multiple log items within a single log statement are joined as one. In case of string concatenation and/or use of non-chastring elements (e.g. variable references, constant references, parameter references, or function calls), all non-charstring elements are currently substituted by an empty string. Note that charstring elements used as parameters in e.g. function calls currently will also be taken into consideration when checking the log statement format.

For individual log items, the **checkLogItemFormat** setting can be enabled instead. The same regular expression is used for checking individual log items. Note that even though the same regular expression is used, both checks are independent, meaning both can be enabled at the same time, with one checking the individual log items within a log statement and the other combining those log items and checking their concatenation (ignoring non-charstring elements as noted above). Due to possible confusion introduced by this additional check and its questionable usefulness, it should be considered deprecated and will be removed in future releases.

By enabling the **processSubsequentLogStatementsAsOne** setting, subsequent log statements can be combined and analyzed as one (again ignoring variables within log statements), in a way similar to how log items are analyzed together within a log statement. This setting has no effect if **checkLogStatementFormat** is disabled.

Note that subsequent log statements (without any other statements in between) are always combined when the **processSubsequentLogStatementsAsOne** setting is enabled. This may in some cases lead to unreliable results, if, for example, one well-formed log statement is followed by another not well-formed one, the combination of the two may still yield a valid log statement with the setting enabled.

Note also if the **processSubsequentLogStatementsAsOne** setting is enabled, the subsequent log statements that are combined together are analyzed as a bundle starting with the first log statement in the sequence, meaning that the subsequent log statements are not analyzed individually again.

Example:

```
module checkLogFormatModule {
    function f_1 ()
    runs on myComponent {
        //incorrect
        log ( "***" )
        test();
        //correct
        log("*** f_1: INFO: OK - random value = " & bit2str(v_random) & " ***");
        test();
        //correct - function calls are ignored
        log("*** f_1: " & getMyStatus() & "INFO: OK - random value = " & bit2str(v_random) & "
        test();
        //incorrect - charstring function call parameters are taken into consideration resulti
        //an incorrect log statement format
        log("*** f_1: " & getMyStatus("1") & "INFO: OK - random value = " & bit2str(v_random)
    }
    const integer c_1 := 1;
    testcase t_sendMsg ()
    runs on myComponent {
                //this should also be correct
                log("*** t_sendMsg: INFO: Unknown component " & p_variable & " ***");
                f_2 ();
        // correct
        log ( "*** t_sendMsg: INFO: Wrong message has been received ***" )
        f_1 (); //if this statement were absent
                //and subsequent log processing enabled
                //the above log sattement will be analyzed
                //together with the following split log statements
        // also correct, given subsequent log processing is enabled
        log ( "*** t_sendMsg: " );
```

```
        log ( "INFO: " );
        log ( "Wrong message has been received ***" );
        f_1 ();
        //correct , given subsequent log processing is enabled
        log ( "*** t_sendMsg: " );
        log ( "INFO: " );
        log ( f_2 () );
        log ( "Wrong message has been received ***" );
        f_1 ();
        // some simple malformed log statements
        //will be combined as one if subsequent log processing is enabled
        log ( "*** : INFORMATION: Wrong message has been received ***" )
        log ( "** t_sendMsg: INFO: Wrong message has been received **" )
        log ( "*** t_sendMsg22: INFO: Wrong message has been received ***" )
        log ( "*** t_sendMsg: INFORMATION: Wrong message has been received ***" )
        //correct under different configuration / not combined with the above
        log ( "*** INFO: Wrong message has been received ***" );
    }
    control {
        //incorrect
        log ( ".." );
        //correct, if not combined with the above
        log ( "*** t_sendMsg22: INFO: Wrong message has been received ***" )
    }
}
```

External Function Invocation Must Be Preceded By A Log Statement

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

Checks whether any invocation of an external function is preceded by a log statement. The examples shall generated warnings for all the invocations that do not have a log statement preceding them directly (tagged as *bad*).

Example:

```
module checkExternalFunctionInvocationFollowedByLogStatement {
        external function fx_example1();
        external function fx_example2(integer p_int) return integer;
        external function fx_example2(charstring p_cs) return charstring;
        //inconclusive - in a constant definition
        const integer c := fx_example1(1);
        //inconclusive - unresolved
        function f_example1() {
                fx_example0();
        }
```

```
        //bad - at the end of a scope
        function f_example1() {
                fx_example1();
        }
        //bad - followed
        function f_example2() {
                fx_example1();
                log("External Function fx_example1 called!")
        }
        //bad - at the end of a scope without SemiColon
        function f_example3() {
                fx_example1()
        }
        //bad - at the end of a scope without SemiColon, followed
        function f_example4() {
                fx_example1()
                log("External Function fx_example1 called!")
        }
        //multiple
        testcase tc1() runs on mtcType system systemType {
                //bad
                fx_example1();
                log("External Function fx_example1 called!")
                f_example1();
                //good
                log("External Function fx_example1 called!")
                fx_example1();
                f_example2()
                //bad
                fx_example1()
                //bad
                fx_example1();
                log("External Function fx_example1 called!")
        }
        control {
                execute(tc1());
        }
}
```

Note that if an external function is called within a constant or a template definition on the module level, an information message will be provided that in such a context it is not possible to have a log statement following it. Note also that if a function definition cannot be resolved, a corresponding information message will be provided as well.

Inconclusive or Fail Setverdict Statement Must be Preceded by a Log Statement

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

Checks whether setverdict statements that set inconc or fail verdicts are preceded by log statements. In the example, the first alt statement represents the expected syntax whereas the second alt statement fails to have log statements before the two existing fail and inconc setverdict statements.

Example:

```
module checkInconcOrFailSetverdictPrecededByLog {
    testcase t_sendMsg() runs on myComponent {
        p1.send(msg_a);
        // this is as expected
        alt {
            [] p2.receive(msg_b) {
                someOtherfunction();
                log("*** t_sendMsg: INFO: Wrong message has been received ***")
                setverdict(fail);
                                someOtherfunction();
            }
            [] p2.receive(msg_c) {
                setverdict(pass);
            }
            [] p2.receive {
                log("*** t_sendMsg: INFO: Unexpected message, possibly malicious ***");
                setverdict(inconc);
            }
        }
        // here, the log statements are missing
        alt {
            [] p2.receive(msg_b) {
                setverdict(fail);
            }
            [] p2.receive(msg_c) {
                setverdict(pass);
            }
            [] p2.receive {
                setverdict(inconc);
            }
        }
    }
}
```

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, aliasInCheckNoOverSpecificRunsOnClauses (since v2.0.0b27)

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{
        }
}
```

As of v2.0.0b27 an additional setting **aliasInCheckNoOverSpecificRunsOnClauses** is provided. It enables the special treatment of component "alias" type definitions (component type definitions without any owned definitions that extend another component type and inherit its definitions). If this setting is enabled (which it is by default), a function or an altstep specified to run on a given component type will only raise a warning if none of the definitions of the component type(s) that this component type extends directly (i.e. not considering the components extended by those components recursively) are used within the function or altstep. It supersedes the previously introduced **extendsInCheckNoOverSpecificRunsOnClauses** setting.

For example:

```
//base component
type component componentWithDefinition {
    timer definedTimer;
}

//extension with own definitions
type component directExtension extends componentWithDefinition {
    var integer directExtensionVariable;
}

//multi alias / extension ("pure", without own definitions)
type component indirectAlias extends directExtension {

}

//multi alias / extension ("pure", without own definitions)
type component pureAlias extends componentWithDefinition {

}

//always good - base component
function someFunctionOnBaseComponent ()
runs on componentWithDefinition {
    definedTimer.start ( 10.0 );
}

//bad - no definition from directExtension used
function someFunctionOnDirectAlias ()
runs on directExtension {
    definedTimer.start ( 10.0 );
```

```
}


//bad - no definition from aliased directExtension used
function someFunctionOnMultiAlias ()
runs on indirectAlias {
    definedTimer.start ( 10.0 );
}

//good - definition from aliased directExtension is used (directAliasVariable)
function someFunctionOnMultiAliasWithReferenceToVariable ()
runs on indirectAlias {
    definedTimer.start ( 10.0 );
    directExtensionVariable := 1;
}

//good - definition from aliased componentWithDefinition is used (definedTimer)
function someFunctionOnPureAlias ()
runs on pureAlias {
    definedTimer.start ( 10.0 );
}
```

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 >
```

The meta-types enumerated, union, record of, record, set of, set, are currently set to be excluded by default when a new configuration profile is generated.

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.

Test Suite Modularization: Module Containment

Modules, whose identifiers contain any of the following substrings, shall contain only certain definition types, permissible for the particular type of module. These substrings will be referred to as "module restrictions" in the following, to establish a unified terminology and simplify the descriptions. All module restrictions are case-sensitive. The configuration entries are following the schema "check{ModuleRestriction}ModuleContainmentCheck", where "ModuleRestriction" is to be substituted with the particular substring. Apart from the permissible definition types, all modules allow the presence of import and group definitions. The output for the quality checks indicates the location/scope of the definition (starting line - end line) violating the particular constraints and the type of module restriction that has been recognized and applied (based on the substring that has been found in the module name). Note that, if a module name contains multiple restrictions, all the checks will be applied individually. This means that a "TypesAndValuesAndTemplates" module will first be checked for the permissible definition types for a "TypesAndValues" module and throw a warning on any other definition (be it a definition of the permissible definitions for a "!Template" module), and then the other way around for the permissible definitions in a "!Templates" module.

Note that control part definitions are allowed in all the following module containment checks. This may be subject to changes in the future (restricted to the "TestControl" modules only).

The details for the individual module restrictions are outlined below.

TypesAndValues Module Must Contain Only Type and Constant Definitions

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

Any module that contains the substring *TypesAndValues* in its name will be analyzed. Only type and constant definitions are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled. The first three definitions are type and constant definitions. However, templates, testcases, and functions must not be present in such a module. Thus, these last three module definitions will be reported as not of the permissible types.

Example:

```
module checkTypesAndValuesModuleContainmentCheckBad {
    import from LibCommon_time all;

    // good part
    type record MyRecordType {
        integer field1 optional,
        charstring field2,
        boolean field3
    }

    type integer myInt;

    const myInt myIntValue := 2;
    // bad part

    template MyRecordType MyTemplate1 := {
        field1 := 123,
        field2 := "A string",
        field3 := true
    }

    testcase t_sendMsg() runs on myComponent {
    }

    function myfunc() {
    }
        group g_1{

                function f_1(){
                }
        }
}
```

Notes: Component and port type definitions also fall under the generic "type definitions" term. Therefore, they are also allowed in a "TypesAndValues" module. This may be subject to changes in the future.

Templates Module Must Contain Only Template Definitions

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

Any module that contains the substring *Templates* in its name will be analyzed. Similar to the "TypesAndValues" module restriction, only template definitions are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled. The first three definitions are template definitions (one is within a group). The constant and function definitions that follow, however, are not allowed

within a "Templates" module and therefore will produce a warning.

Example:

```
module checkTemplatesModuleContainmentBad {
    import from LibCommon_time all;

    // good part, only templates
    template MyRecordType t_1 := {
        field1 := 1,
        field2 := "a",
        field3 :=true
    }

    template integer t_2 := (0,1,2,3,4,5,6,7);


    group GroupedDefs{
            template integer t_3 :=(0 .. 10);
                //bad part
                const integer c_1 := 2;

    }

        //bad part
        function f_2(){
        }

}
```

Functions Module Must Contain Only Function and Altstep Definitions

- **Symbolic Name in XML Configuration**: checkFunctionsModuleContainmentCheck
- **Dependant Tags in XML Configuration**: checkFunctionsModuleContainmentCheckAllowExtFunction

Any module that contains the substring *Functions* in its name will be analyzed. Similar to the other module restrictions, only function and altstep definitions are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled. Additionally, external function definitions may (by default) or may not be allowed within such modules, depending on whether **checkFunctionsModuleContainmentCheckAllowExtFunction** is enabled or not.

Example:

```
module checkFunctionsModuleContainmentBad {
    import from LibCommon_time all;
```

```
    // good part, only functions and altsteps
    function f_1 () {
    }
    altstep a_1 () {
    }
    //bad part
    type record typeA {
        integer field1,
        boolean feild2
    }
    group GroupedDefs {
        function f_2 () {
        }
        altstep a_2 () {
        }
        const integer c_1 := 1;
    }
    //configurable part
    //depending on the configuration external functions
    //may (default) or may not be allowed
    external function ef_1 ();
}
```

Testcases Module Must Contain Only Testcase and Function Definitions That Are Referenced In Start Statements

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

Any module that contains the substring *Testcases* in its name will be analyzed. Similar to the other module restrictions, only testcase and function definitions that are referenced in start statements are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled.

Example:

```
module checkTestcasesModuleContainmentBad {
    import from LibCommon_sync all;

    // bad module
        function f_1() runs on component_1{
        }

        testcase tc_1() runs on component_1{
                        var component_1 ptc0 := component_1.create;
                        ptc0.start(f_1());
                        ptc0.start(f_0()); // defined elsewhere
```

```
                        f_3(); //does not count - not in a start statement
        }
        //not used in a start statement
        function f_3() runs on component_1{
        }

    //not a testcase or a function\
    const integer c_1 := 0;


    group GroupedDefs{
        //not used in a start statement
                function f_4() runs on component_1 return float{
                }

                function f_2() runs on component_1{
                        var component_1 ptc1 := component_1.create;
                        ptc1.start(f_1());
                        ptc1.start(f_0(f_4())); // does not count - used as a nested parameter
                        f_4(); //does not count - not in a start statement
                        timer t;
                        t.start(f_4()); //does not count - used in a timer start operation

                }

                testcase tc_2() runs on component_1{
                        var component_1 ptc2 := component_1.create;
                        ptc2.start(f_1());
                        ptc2.start(f_0()); // defined elsewhere

                }

    }
}
```

ModuleParams Module Must Contain Only Modulepar Definitions

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

Any module that contains the substring *ModuleParams* in its name will be analyzed. Similar to the other module restrictions, only modulepar definitions are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled.

Example:

```
module checkModuleParamsModuleContainmentBad {
    import from LibCommon_time all;

    // good part, only module parameters
        modulepar integer mp_1;

        modulepar {
                integer mp_2;
                charstring mp_3;
        }

    //bad part
    const integer c_1 := 1;

    group GroupedDefs{
                modulepar integer mp_4;
                const integer c_2 := 2;
    }
}
```

Interface Module Must Contain Only Component, Port, and Type Definitions

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

Any module that contains the substring *Interface* in its name will be analyzed. Similar to the other module restrictions, only component, port, and type definitions are permitted within such a module (or generally speaking any type definition). The example below illustrates a module which will produce warnings if this quality check is enabled.

Example:

```
module checkInterfaceModuleContainmentBad {
    import from LibCommon_time all;

    // bad module

        type component component_1 {
        }

        type port port_1 message{
                in integer
        }
        const integer c_1 := 0;

        group g_1{
```

```
                type component component_2 {
                }
                type port port_2 message{
                        out integer
                }

                const integer c_2 := 1;
        }

        //are control parts permissible?
        control {

        }

}
```

TestSystem Module Must Contain Only Component and Port Definitions

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

Any module that contains the substring *TestSystem* in its name will be analyzed. Similar to the other module restrictions, only component type definitions are permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled.

Example:

```
module checkTestSystemModuleContainmentBad {
    import from LibCommon_time all;

    // bad module, port definitions as well

        type component component_1 {
        }

        type port port_1 message{
                in integer
        }

        group g_1{
                type component component_2 {
                }
                type port port_2 message{
                        out integer
                }
```

```
        }

        //are control parts permissible?
        control {

        }

}
```

TestControl Module Must Contain Only Control Part Definition

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

Any module that contains the substring *TestControl* in its name will be analyzed. Similar to the other module restrictions, only control part definition is permitted within such a module. The example below illustrates a module which will produce warnings if this quality check is enabled.

Example:

```
module checkTestControlModuleContainmentBad {
    import from LibCommon_time all;

    //bad part

        modulepar integer mp_1;

        modulepar {
                integer mp_2;
                charstring mp_3;
        }


    const integer c_1 := 1;

    group GroupedDefs{
                modulepar integer mp_4;
                const integer c_2 := 2;
    }

        control {

        }

}
```

Test Suite Modularization: Importing Libraries

TypesAndValues Modules Must Always Import From Certain Modules

- **Symbolic Name in XML Configuration**: checkTypesAndValuesModuleImportsLibNames
- **Dependant Tags in XML Configuration**: typesAndValuesImportsLibNamesRegExp, typesAndValuesImportsLibNamesExcludedRegExp

The check inspects whether a module that contains *TypesAndValues* (case-sensitive) in its name imports from modules with certain names (e.g. libraries - *LibCommon*). The names of the required imports are specified by means of regular expressions. The default setting is *.*?LibCommon.**, meaning that definitions shall be imported from modules that contain *LibCommon* in their name (regardless of the position of this substring). This setting is stored under the **typesAndValuesImportsLibNamesRegExp** tag in the configuration.

Additionally, with the help of the **typesAndValuesImportsLibNamesExcludedRegExp** setting, certain modules can be excluded from this check (e.g. modules whose identifier also contains *LibCommon*, such as *!LibCommon_TypesAndValues*). The default value for this setting is *.*?LibCommon.**.

The message text in the output for this quality check is: `Required imports ("$typesAndValuesImportsLibNamesRegExp") not found!`

Testcases Module Must Always Import From Modules Prefixes with !LibCommon_Sync

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

The check inspects whether a module that contains *Testcases* (case-sensitive) in its name imports from modules that are prefixed with the string *!LibCommon_Sync* (case sensitive!).

Similar conditions as in "TypesAndValues Module Must Always Import From Modules Prefixes with LibCommon" apply to this quality check as well. Modules whose identifiers contain "!LibCommon_Sync" are excluded (regardless of where the substring is placed). The configuration for this quality check will also be extended in the future to allow customized selection of the required prefix and the exclusion criteria.

Module Size

- **Symbolic Name in XML Configuration**: checkModuleSize
- **Dependant Tags in XML Configuration**: maximumAllowedModuleSizeInBytes

This check tests whether modules exceed a given reference size (specified by the **maximumAllowedModuleSizeInBytes** configuration entry). The size as the tag suggests specifies the maximum allowed module size in bytes. The default reference size is 10000 bytes (~10KB).

Note that the module size is the subject of analysis and not the file size (accounting for the fact that there may be multiple modules defined within the same file as well). Thus, this quality check may not be violated, even if a file exceeds the maximum allowed size, if there are multiple smaller modules defined within that file, which do not themselves violate the maximum allowed size constraint.