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);
            }
        }
    }
}
```