An introduction to the SuperTest MISRA suites

with examples

The SuperTest MISRA suites are created to verify the conformance of MISRA checking software. The aim of a, so-called, *MISRA checker* is to check application software for its compliance with the MISRA guidelines. The SuperTest MISRA suites in turn verify that such a MISRA checker generates the right diagnostics and not too many false positives or false negatives.

The MISRA suites contain two types of tests: those which a name starting with an 'x', and those which a name that starts with a 't'.

T-tests

The so-called *t-tests* are fully conforming with the MISRA guidelines and should not trigger any diagnostics in the MISRA checking software.

This is an example of a t-test:

```
/*
 * Test for MISRA-C:2004 Rule 4.2:
 *     Trigraphs shall not be used
 */

typedef signed int int32_t;

static int32_t fun (void);
static int32_t fun (void) {
    char r[50] = "OK: ??? ??.??+??~??^"; /* Compliant */

    return (r[0] != 'O') ? (int32_t)1 : (int32_t)0;
}

int32_t main (void) {
    return fun ();
}
```

In this test, the string that initializes the variable `r[50]` contains a number of character sequences that look like trigraph sequences, which the MISRA guidelines do not allow, but are not trigraphs. Hence, this code is compliant with MISRA Rule 4.2. No diagnostic should be given. The code is also constructed so that no other MISRA rules are violated.

X-tests

The *x-tests* contain a violation of one of the MISRA rules. When passed to a MISRA checker, they should trigger a diagnostic for a specific rule.

Here is an example of an x-test that violates Rule 5.4, which requires that tags are unique.

```
/*
 * Test for MISRA-C:2004 Rule 5.4:
 *     A tag name shall be a unique identifier
 *
 * Notes on test:
 * Check that the following code violates MISRA Rule 5.4
 */

typedef unsigned short uint16_t;
typedef signed int int32_t;
typedef unsigned int uint32_t;

int32_t main (void) {
    uint32_t aval;
    {
        enum thetag { x, y, z };
        enum thetag ty = y;
        aval = (ty < x) ? (int32_t)1U : (int32_t)0U;
    }
    {
        struct thetag {   /* Not compliant */
            uint16_t a;
            uint16_t b;
        } str = {3U, 4U};
        aval += str.a;
    }
    return (aval != 3U) ? (int32_t)1 : (int32_t)0;
}
```

Note that this x-test is perfectly fine C90 code. C90 allows the reuse of **`thetag`** in different scopes. But MISRA does not.

A big effort has been put into making the x-tests as specific as possible, so that they contain only one rule violation, and at the same time keep them concise and understandable. This is not always possible. Sometimes rules partially overlap and an additional diagnostic is triggered.

Undecidable Rules

Some of the MISRA rules are what is called *undecidable*. This means that for complex examples of such code, a MISRA checker may not be able to follow the control and data flow with sufficiently detail to decide if a rule is violated or not. For such rules, there are often 'trivial' cases for which a MISRA checker is expected to give a diagnostic. We have tried to include such trivial cases in the suites as well. We have also tried to include more complex cases that are decidable with some effort. The following is an example of a t-test for an undecidable rule, so it should not give a diagnostic:

```
/*
 * Test for MISRA-C:2004 Rule 17.3:
 *     >, >=, <, <= shall not be applied to pointer
 *     types except where they point to the same array
 *
 * Notes on test:
 * Check that the code does not violate MISRA Rule 17.3
 */

typedef signed int int32_t;

int32_t main (void) {
    int32_t a[] = {1, 2, 3};
    int32_t *pa = &a[1];
    int32_t r = 9;

    if (&a[2] > pa) {   /* Compliant */
        r = 6;
    }

    return r - 6;
}
```

To verify that the pointer comparison is compliant, requires that a MISRA checker applies non-trivial data flow analysis to the code. If the MISRA checker cannot do that, it might decide to play it safe and generate a *false positive*: an incorrect diagnostic for a (possible) rule violation. MISRA checker may offer options to suppress false positives.

The MISRA test suite comes in several flavors. By means of (advisory) Rule 6.3, MISRA forces application developers to choose a data model and express that choice in the primitive types used for arithmetic. This can be seen in the examples above, which use the 32-bit integer type "int32_t".

The MISRA suites are supplied both in a generic form and in a form that is specific for a data model. The examples above are all for a specific data model. In the generic form, the previous example looks like this:

```
int main (void) {
    int a[] = {1, 2, 3};
    int *pa = &a[1];
    int r = 9;

    if (&a[2] > pa) {   /* Compliant */
        r = 6;
    }

    return r - 6;
}
```

In this form, one should tell the MISRA checker that it should ignore Rule 6.3.

Solid Sands can also provide additional instances of the MISRA suites that are specific for another data model, if needed.
