

Checking Code and Models in Production Environments

By: Tom Erkkinen and Damon Hachmeister, The MathWorks

Production software development teams routinely perform code inspections, often with the aid of static analysis automation tools. These engineers know that having an automated process for building, inspecting, and checking software is one of the best ways to improve product quality. As such, production processes and tools are evolving and adapting as these companies now migrate to a model based development environment with production code generation.

This article describes methods, tools, and interfaces available from The MathWorks that help automate the checking of models. It also describes how to integrate a Lint tool into the Real-Time Workshop Embedded Coder build process for checking the generated code. An example will be shown that uses Simulink APIs and Real-Time Workshop Embedded Coder *hook* mechanisms to check both the model and its generated C code. The checks are based on MISRA-C, a popular C language safe subset.

Keep in mind while reading this that static analysis complements the numerous dynamic analysis and simulation capabilities provided with Simulink and Stateflow. Developers should decide for themselves what aspects of static and dynamic analysis yield the best results for their development environments. Readers may want to familiarize themselves with the [Targeting Tips](#) article presented in the May 2003 issue of MATLAB Digest. This article provides background information regarding details of the new *uset_param/uset_param* APIs, as well as, the *ert_make_rtw_hook_file*, both of which are applied within.

MISRA-C

The Motor Industry Software Reliability Association first published the “Guidelines For The Use Of The C Language In Vehicle Based Software” in April 1998. MISRA-C, as it is commonly known, was created in large part by automotive industry developers who embed C in their electronic control unit (ECU) software. A total of fifteen vehicle manufactures and suppliers are credited in the MISRA-C document as contributors. However, MISRA-C is also used in other industries, including aerospace and defense, due to the pervasiveness of C as the language of choice for developing embedded systems. See the [MISRA web site](#) for more information on the MISRA organization and its other software engineering documents and standards.

MathWorks MISRA-C Compliance Package

MathWorks products are increasingly used to generate embedded software and applications. As a result, many production users have asked about the level of MISRA-C compliance for the Real-Time Workshop Embedded Coder. The MathWorks has recently created a MISRA-C compliance analysis package to address this. This package is based on model inspections, code analysis, and quality engineering product tests suites. This package assessed MathWorks Release 13 but was a follow-on to an earlier inspection-based approach done for Release 12.1.

The MISRA-C compliance package includes:

- Overview document
- Compliance matrix
- Presentation of violations and mitigations
- Simulink and Stateflow model examples
- Real-Time Workshop Embedded Coder code examples

This package is not an exhaustive analysis. It does, however, represent the current understanding by The MathWorks of the level of MISRA-C compliance for its code-generation products. This was an organizational effort with contributions from diverse departments, including product development, quality engineering, technical marketing, and documentation.

The MISRA-C compliance package is available upon request to our production code user community for input, review, and assessment. It is to be considered a work in progress.

The discussion that follows describes how users can develop, integrate, and apply static analysis throughout the development process. It starts with an approach for integrating a Lint analysis tool and then describes how to develop and incorporate model checks using MathWorks APIs. Motivations for using both code and model checkers are discussed throughout. One of the MISRA rules is used as the working example.

SPLINT

According to the [Splint web site](#), *Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint.*

Splint does not have checks built in specifically for MISRA, but nonetheless, it does serve our purpose of demonstrating how to integrate a code analysis tool into a build environment. Splint is licensed under the GNU General Public License (GPL) and is available at no cost. It also supports the Windows platform, furthering its usefulness as a demonstration tool for this article.

The [MISRA Web site](#) lists a number of commercially available products from companies such as [LDRA](#) that specifically check code for MISRA-C compliance. Some of these tools provide other capabilities, including code compilation, static analysis, flow graphing, and structural coverage. Readers are welcome to substitute these or other tools for the Splint example that follows.

Splint Installation procedure (for Windows):

1. Download Splint from www.splint.org (version 6.0.1 was used for this article)
2. Unzip the downloaded zip file into the directory where you want to install *splint*
3. Set environment and path variables LARCH_PATH and LCLIMPORTDIR as described in installation procedures

4. Run test models in test directory by invoking *splint* and verifying the results

Using Splint to Check for MISRA-C

Although Splint does not check specifically for MISRA, it is a Lint tool and as such, has a number of settings that can be used to check for many of the items called out in MISRA. One example is the setting `-internal-name-length value`. It checks identifiers for uniqueness based on the number of characters, or *value*, supplied by the user. By setting the *value* to 31, it will check for the following Rule described in the MISRA-C document.

MISRA Rule 11 (required): *Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.*

This rule is important in order to avoid name clashes with identifiers that have long names. This rule also makes code more portable since most compilers and linkers support at least 31 characters of significance.

The following example shows a hand-written C code violation example. It is followed by the specific *splint* invocation command and the generated output that detects and reports the violation.

Hand Code Violation Example for filename test1.c

```
#include <stdio.h>
int main(void)
{
    int abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz1 = 1;
    int abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz2 = 2;
    printf("\n first value = %d \n", abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz1);
    printf("\n second value = %d \n", abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz2);
}
```

Splint Invocation and Output:

```
%splint -weak -nolib -internal-name-length 31 test1.c
```

```
Splint 3.0.1.6 --- 11 Feb 2002
```

```
test1.c: (in function main)
```

```
test1.c(5,6): Internal identifier abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz2 is not distinguishable from abcdefghijklmnopqrstuvwxyz_abcdefghijklmnopqrstuvwxyz1 in the first 31 characters (abcdefghijklmnopqrstuvwxyz_abcd). An internal name is not distinguishable from another internal name using the number of significant characters. According to ANSI89 Standard (3.1), a implementation may only consider the first 31 characters significant (ISO C99 specified ...
```

```
Finished checking --- 1 code warning
```

Note that

- `-weak` limits the warnings reported that are not related to the specified checks.
- `-nolib` restricts the inspection to just the file and does not load standard libraries.

Readers are encouraged to assess these and other *splint* options for their own purposes.

Using Built-in Model Settings to Check for MISRA-C

A simple Simulink model is shown below. As you can see, this model uses long names for its input signals and atomic subsystems. This will result in long names for the input variables and functions in the generated code.

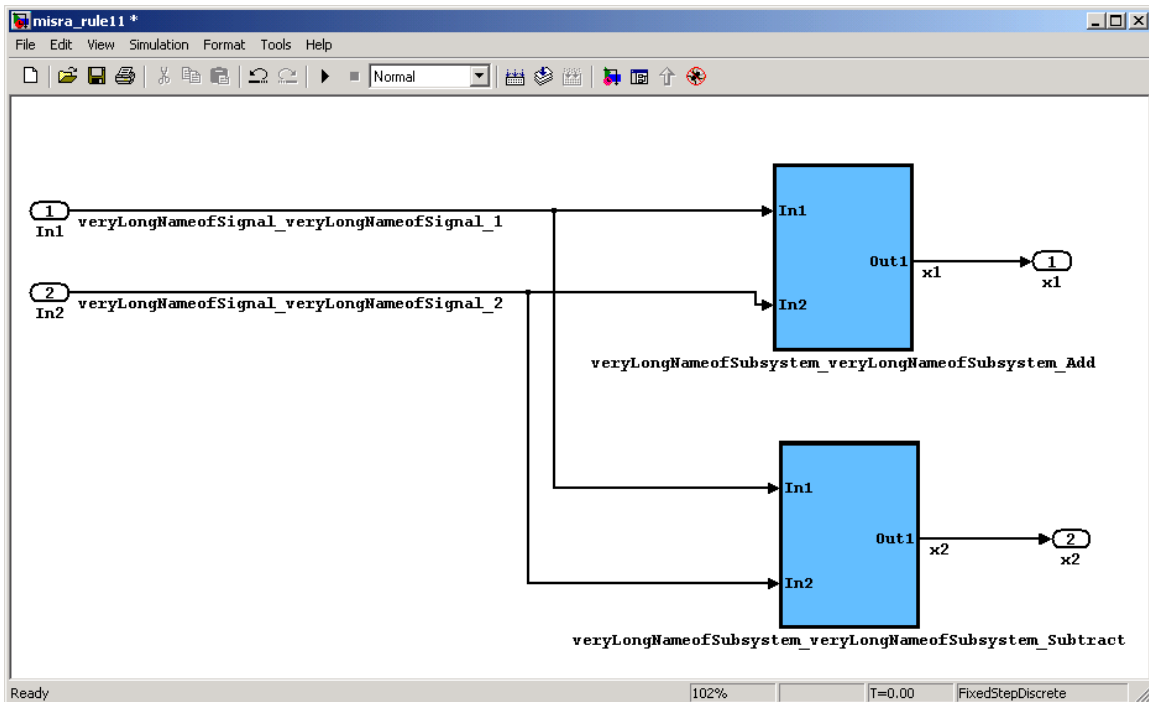


Figure: Simulink model (misra_rule11.mdl) with lengthy identifiers

Code from this model was generated using one of the newly optimized system target files that are now available for the Real-Time Workshop Embedded Coder. These targets set the various code generation configuration options to values that will generally yield the most optimized code. Two versions exist: one optimized for fixed point and another for optimized floating point. The floating point target was used here, as shown below.

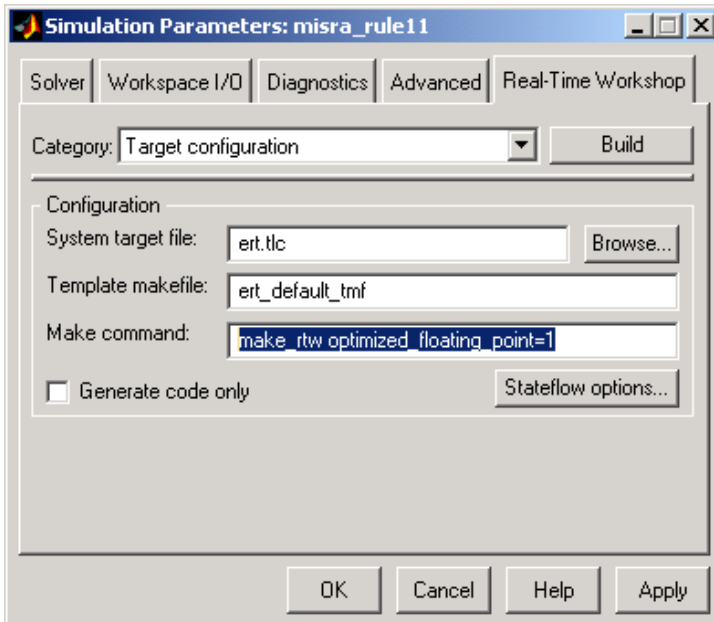


Figure: New optimized Real-Time Workshop Embedded Coder system target

These new targets are available by request from The MathWorks.

The storage classes of the output signals (x1 and x2) were set to External Global. The storage classes of the input signals (the ones with the long names) were set to Simulink Global. For conciseness, the option to generate comments ('GenerateComments'), which helps trace the code to the model, was turned off. The maximum identifier length ('MaxIdLength') was left at its default value of 31 as shown below.

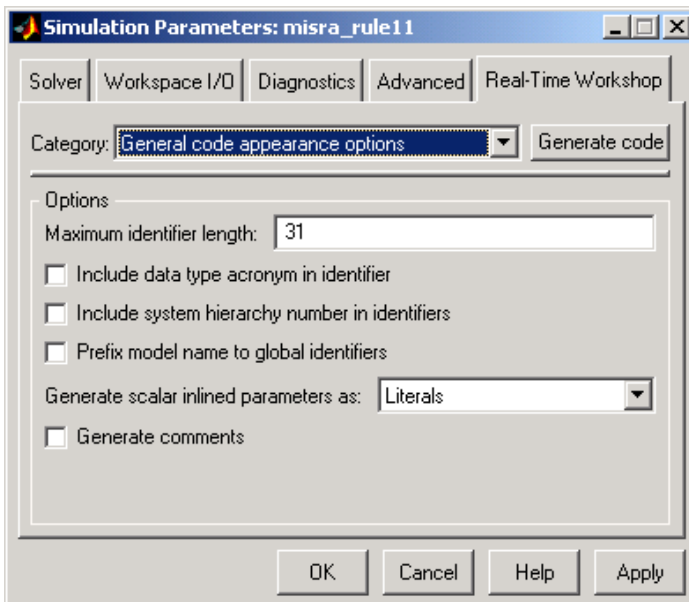


Figure: Code generation settings with identifiers limited to 31 characters

MISRA-C Compliant Example

Review the resulting automatically generated code below and note that the lengthy identifier names in the model were abbreviated in the code based on the `MaxIdLength` setting of 31 characters. A name modification scheme was employed to ensure that the generated identifiers do not clash. This results in the `_a` and `_b` suffixes for the variables and functions that map to the corresponding signal and subsystem names. This code complies with Rule 11 of MISRA-C.

Automatically Generated Code for misra_rule11.mdl – MISRA Compliant

```
#include "misra_rule11.h"
#include "misra_rule11_private.h"

real_T x2;
real_T x1;

ExternallInputs rtU;

void veryLongNameofSubsystem_a(void)
{
    x1 = rtU.veryLongNameofSignal_veryLong_a
        + rtU.veryLongNameofSignal_veryLong_b;
}

void veryLongNameofSubsystem_b(void)
{
    x2 = rtU.veryLongNameofSignal_veryLong_a
        - rtU.veryLongNameofSignal_veryLong_b;
}

void misra_rule11_step(void)
{
    veryLongNameofSubsystem_a();
    veryLongNameofSubsystem_b();
}
```

MISRA-C Violation Example

The storage classes of the input signals (i.e., the ones with the long names) will now be set to `ImportedExtern`. As a result, the maximum identifier length processing within the Real-Time Workshop Embedded Coder will not be applied because these are imported signals and variables that were produced outside the current system's boundary.

The maximum identifier length setting is then set to 100 characters. This setting will result in the generation of the full subsystem names as shown below.

Automatically Generated Code for misra_rule11.mdl – MISRA Violation

```
void veryLongNameofSubsystem_veryLongNameofSubsystem_Add(void)
{
    x1 = veryLongNameofSignal_veryLongNameofSignal_1
        + veryLongNameofSignal_veryLongNameofSignal_2;
}

void veryLongNameofSubsystem_veryLongNameofSubsystem_Subtract(void)
```

```
{
  x2 = veryLongNameofSignal_veryLongNameofSignal_1
    - veryLongNameofSignal_veryLongNameofSignal_2;
}
```

The code from the Compliant Example shows that one can satisfy the MISRA rule and generate compliant code by using a built-in Real-Time Workshop setting (MaxIdLength = 31). However, code from the Violation Example shows that code can be generated from MathWorks products that violates the MISRA rule.

This examples illustrates two main sources of potential MISRA-C violations that need to be checked for within a model based development environment:

- Importing or interfacing to legacy code that violates a rule(s)
- Conscious or accidental changes to built-in Real-Time Workshop setting(s)

For these reasons and others, it makes sense to add a code based checker to your model based development environment, which brings us to the topic of how to integrate Splint into the build process.

Integrating Splint and Real-Time Workshop Embedded Coder

The procedure for integrating tools and automatically configuring the code generator at various points during the build process is straightforward and clearly documented. The general approach was described in the [Targeting Tips](#) article presented in the May 2003 issue of MATLAB Digest. That discussion will not be repeated here.

The specific change that is needed within the `ert_make_rtw_hook` M-file to integrate *splint* is shown in the code below. Note that the choice of the hook entry point was the Before Make entry point, which occurs after the code was generated but before the compile and link procedures begin.

ert_make_rtw_hook with Splint Integration (shown in bold)

```
case 'before_tlc'
  % Called just prior to invoking TLC Compiler (actual code generation.)
  % Valid arguments at this stage are hookMethod, modelName, and
  % buildArgs

case 'before_make'
  % Called after code generation is complete, and just prior to kicking
  % off make process (assuming code generation only is not selected.) All
  % arguments are valid at this stage.
  system(['splint -weak -nolib -internal-name-length 31 ', modelName, '.c']);

case 'exit'
  % Called at the end of the RTW build process. All arguments are valid
  % at this stage.
```

Code is now generated using the new `ert_make_rtw_hook` file. Splint now detects the violations and displays them within the MATLAB command window that follows.

```

MATLAB
File Edit View Web Window Help
Current Directory: D:\MatlabR1301_LCS\work

### Writing header file misra_rule11_types.h
### Writing source file misra_rule11.c
.
### Writing header file misra_rule11_private.h
### Writing source file ert_main.c
### TLC code generation complete.
Splint 3.0.1.6 --- 11 Feb 2002

misra_rule11_private.h(60,15): Internal identifier veryLongNameofSignal_veryLong
NameofSignal_2 is not distinguishable from veryLongNameofSignal_veryLongName
ofSignal_1 in the first 31 characters (veryLongNameofSignal_veryLongNa)
An internal name is not distinguishable from another internal name using the
number of significant characters. According to ANSI09 Standard (3.1), an
implementation may only consider the first 31 characters significant (ISO C99
specified 63). The +internalnamelen <n> flag changes the number of
significant characters, -internalnamecaseinsensitive to makes alphabetical
case significant, and +internalnamelooklike to make similar-looking
characters non-distinct. (Use -distinctinternalnames to inhibit warning)
misra_rule11_private.h(59,15): Declaration of veryLongNameofSignal_veryLongNa
meofSignal_1
misra_rule11_private.h(63,13): Internal identifier veryLongNameofSubsystem_veryL
ongNameofSubsystem_Subtract is not distinguishable from
veryLongNameofSubsystem_veryLongNameofSubsystem_Add in the first 31
characters (veryLongNameofSubsystem_veryLon)
misra_rule11_private.h(62,13): Declaration of veryLongNameofSubsystem_veryLon
gNameofSubsystem_Add

Finished checking --- 2 code warnings
### misra_rule11.mk which is generated from D:\MatlabR1301_LCS\rtw\c\ert\ert_vc.tmf is up to date
>>

```

Figure: Splint reporting violations within MATLAB

It should now be clear that it is possible to detect MISRA violations in the code generated by Real-Time Workshop Embedded Coder by integrating lint or a commercially available MISRA tool into the build process. However, software projects are best served by detecting and eliminating errors as soon as possible. This brings us to the topic of model checking.

Developing Custom Model Scripts to Check for MISRA-C

MathWorks has numerous APIs that can be employed to develop sophisticated model checking capabilities. In fact, third-party connection partner tools such as [Mint](#) exist, which leverage these core capabilities. MathWorks APIs access many items or objects within Simulink and Stateflow. These items include model and code-generation settings.

Overview of APIs

Here are some of the foremost APIs that developers should acquaint themselves with:

- `find_system` – Obtains the Simulink objects in your model including systems, subsystems, and blocks
- `sfroot` – Obtains the Stateflow root object
- `get_param/set_param` - Accesses Simulink objects and their parameters
- `uget_param/uset_param` - Accesses Simulink and Stateflow code generation params

It is also useful to know short cuts for obtaining object names:

- `bdroot` – Name of the top level Simulink system

- gcs – Gets name of current (selected) Simulink system
- gcb – Gets name of current (selected) Simulink block

Using APIs

The following example uses the APIs to detect requirements not yet satisfied in a model. It will be assumed that the model guidelines for the example require that each block used to satisfy a specified requirement have its Tag block property labeled as REQ (for requirement). Also, the guidelines require that you use the Description block property to identify each block tagged with a requirement as either Open or Closed. The following two figures show the block property forms for blocks with an open requirement and with a closed requirement.

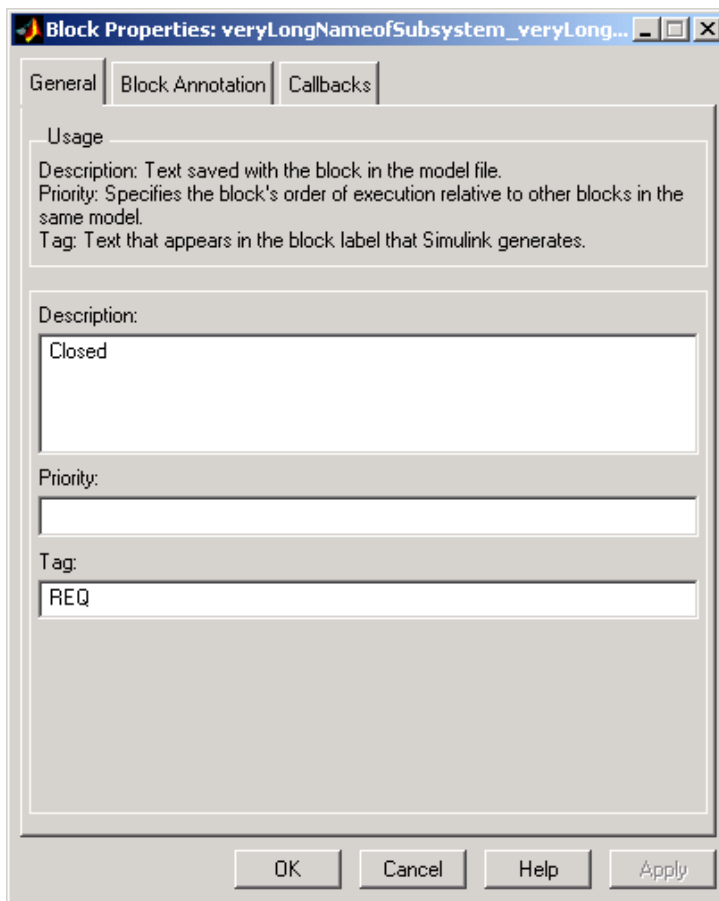


Figure: Block Properties – Closed REQ

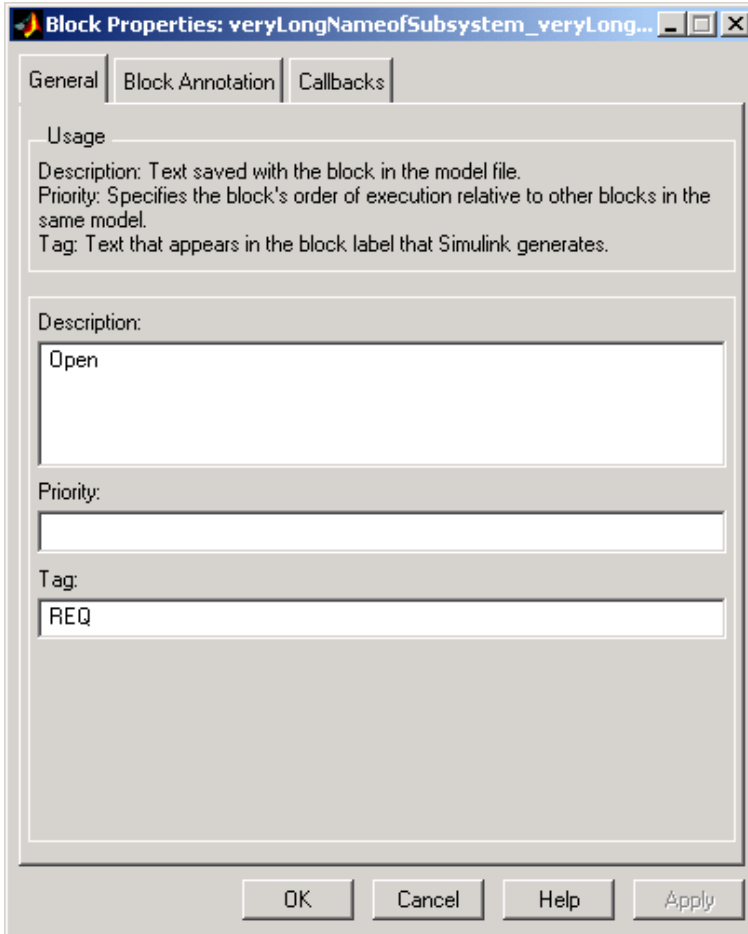
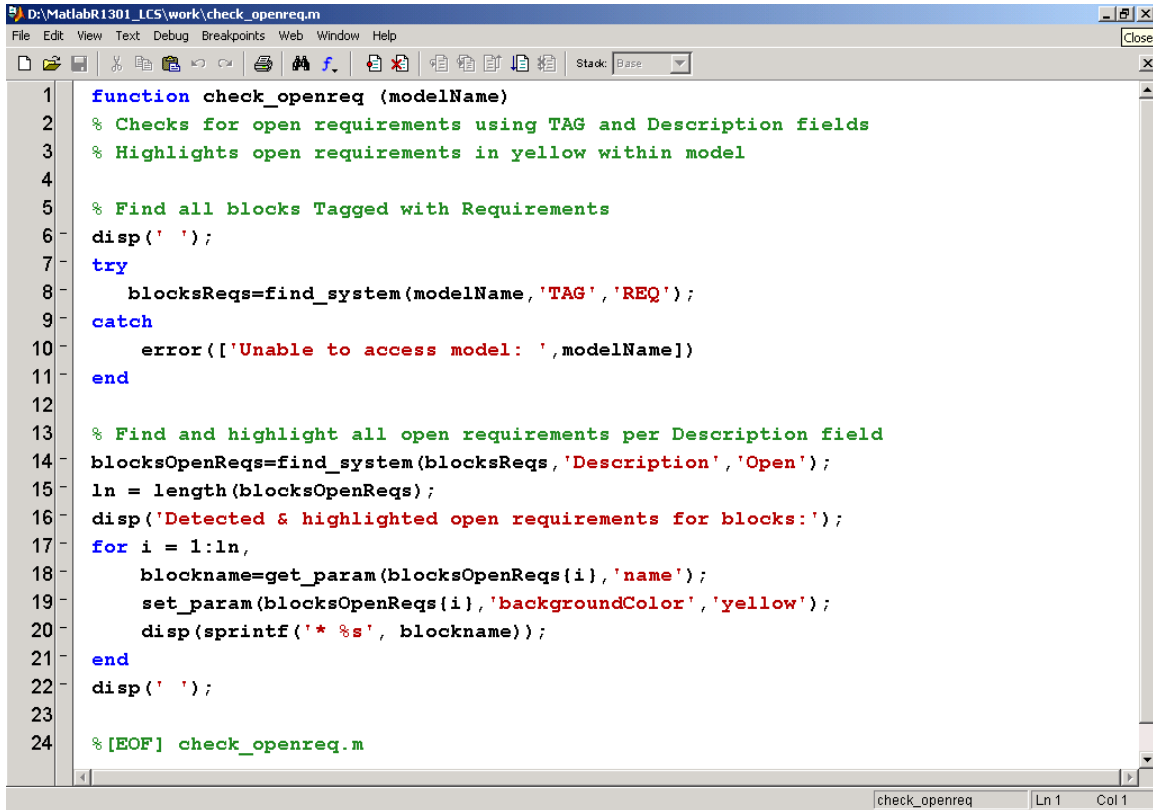


Figure: Block Properties – Open REQ

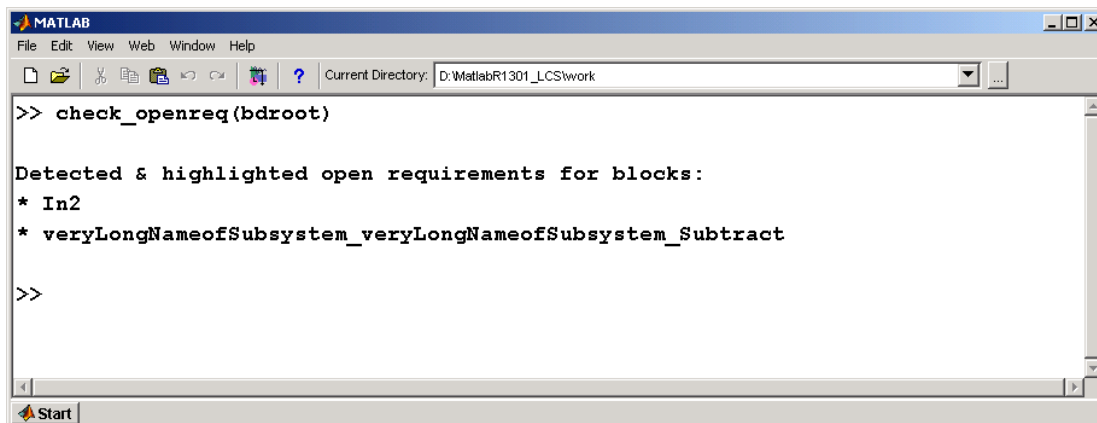
A series of MATLAB commands is developed based on the Simulink APIs to identify blocks with the Open requirements. It then highlights them in yellow within the model. The commands were placed in *check_openreq.m* script file shown below.



```
1 function check_openreq (modelName)
2 % Checks for open requirements using TAG and Description fields
3 % Highlights open requirements in yellow within model
4
5 % Find all blocks Tagged with Requirements
6 disp(' ');
7 try
8     blocksReqs=find_system(modelName, 'TAG', 'REQ');
9 catch
10     error(['Unable to access model: ', modelName])
11 end
12
13 % Find and highlight all open requirements per Description field
14 blocksOpenReqs=find_system(blocksReqs, 'Description', 'Open');
15 ln = length(blocksOpenReqs);
16 disp('Detected & highlighted open requirements for blocks:');
17 for i = 1:ln,
18     blockname=get_param(blocksOpenReqs{i}, 'name');
19     set_param(blocksOpenReqs{i}, 'backgroundColor', 'yellow');
20     disp(sprintf('* %s', blockname));
21 end
22 disp(' ');
23
24 % [EOF] check_openreq.m
```

Figure: check_openreq.m MATLAB file

The *check_openreq.m* file was executed from within MATLAB for a given Simulink diagram. Two blocks were identified as having open requirements and the two blocks were then highlighted in yellow as shown in the following two figures.



```
>> check_openreq(bdroot)

Detected & highlighted open requirements for blocks:
* In2
* veryLongNameofSubsystem_veryLongNameofSubsystem_Subtract

>>
```

Figure: Invoking check_openreq.m within MATLAB

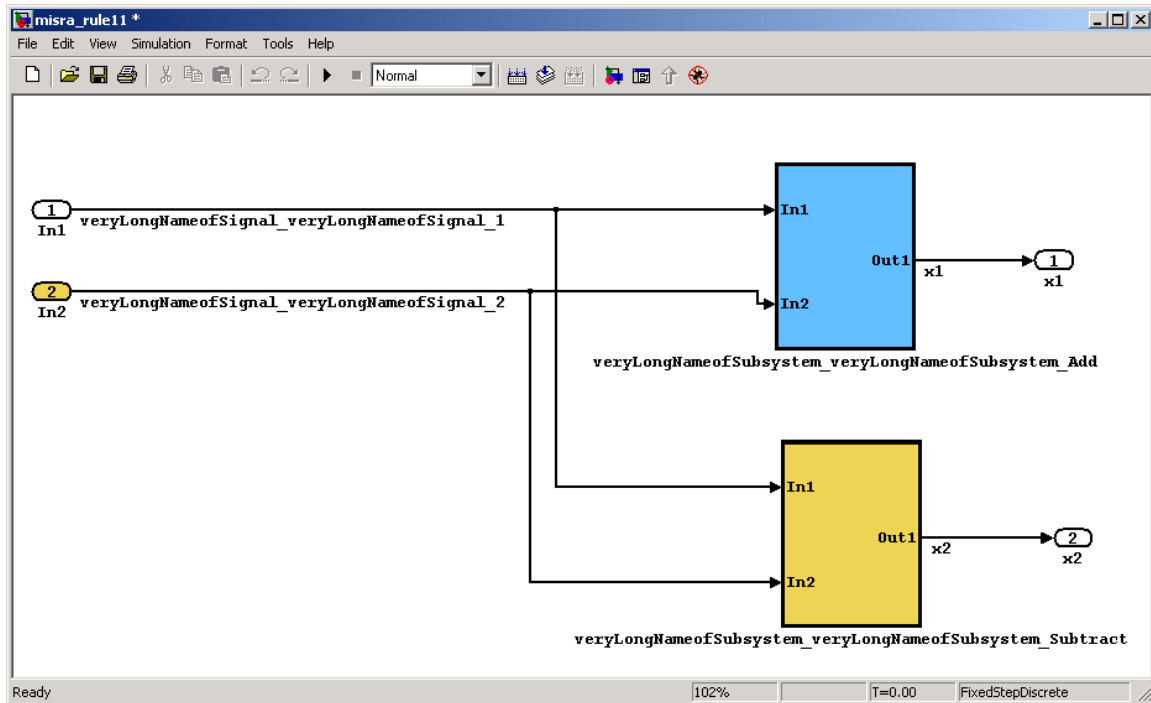


Figure: Two highlighted blocks with open requirements

Another useful MATLAB command is `inspect`. It opens the property inspector based on a supplied object handle. The property inspector lets users interactively change the properties of the supplied object. It is not always apparent which properties are accessible or exactly what they do, making the inspector a useful design aid.

One could also get this list of properties on the selected subsystem block by issuing the following MATLAB command:

```
>> get_param(gcb, 'ObjectParameters')
```

To invoke the inspector on the selected subsystem, issue the following MATLAB commands:

```
>> hdl = get_param(gcb, 'Handle');
>> inspect(hdl)
```

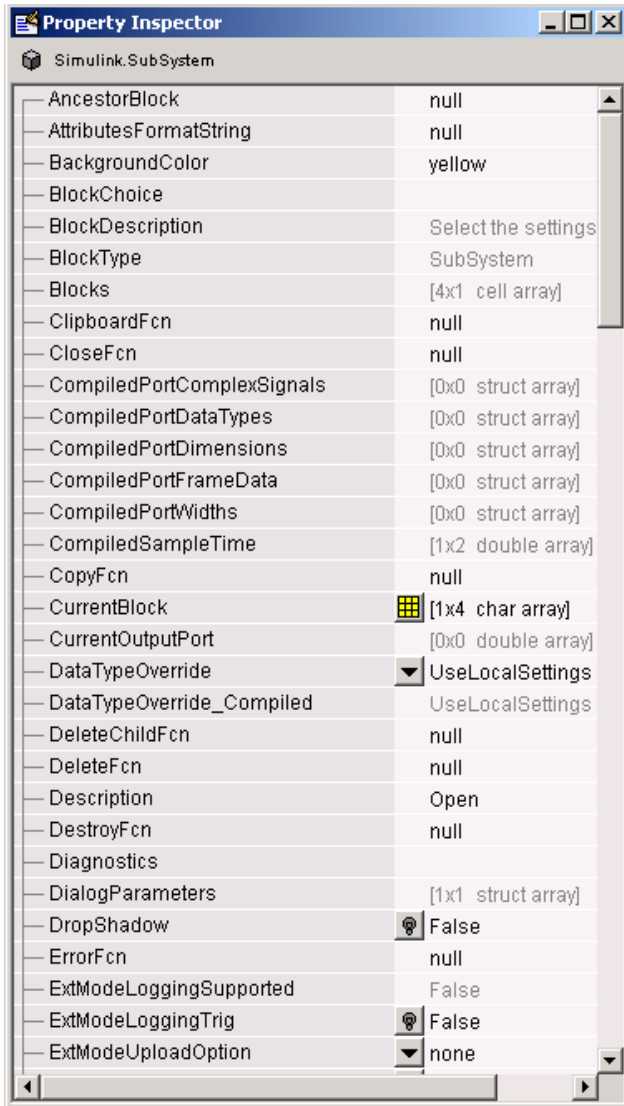


Figure: Inspector for open requirement subsystem

Applying Model Checker

We will next review how to apply MISRA checks to a model now that a few of the key Simulink APIs have been described. The first step is to write a MATLAB file (M-file) that contains the MATLAB commands, or scripts, that check a model for MISRA compliance. Next, we will modify the `ert_make_rtw_hook` file and incorporate the script at a point early on during the build process, prior to code being generated.

The script that accomplishes the first step (checker) is shown below.

```
1 function check_misra (modelName)
2 % check_misra- Configuration file that checks a model for MISRA compliance
3 % Copyright 1984-2003 The MathWorks, Inc.
4 % $Revision: 1.5 $ $Date: 2003/06/15 21:45:07 $
5
6 disp(sprintf('\n\n Checking model for MISRA-C compliance', modelName));
7 % catch error for first model check
8 try
9     ver = uget_param(modelName, 'ModelVersion');
10 catch
11     error(['Unable to access model: ', modelName]);
12 end
13
14 % display model name and version number
15 disp(sprintf('Model Name: %s \nVersion: %5s \n', modelName, ver));
16
17 % Check Code Generation Max Identifier Setting based on MISRA Rule 11
18 maxid = uget_param(modelName, 'MaxIdLength');
19 if (maxid > 31)
20     disp('POSSIBLE MISRA RULE 11 VIOLATION - Maximum id length is set greater than 31');
21 end
22
23 % Display and check model name length as noted in MathWorks MISRA Compliance Matrix
24 modelIdLength=length(modelName);
25 if (modelIdLength > 19)
26     error(['MISRA RULE 11 VIOLATION - The model name (' , modelName, ') has ', int2str(modelIdLength), ' chars.', ...
27         ' Model names should be less than 20 chars since they are prepended to internal structures']);
28 end
29
```

Figure: MATLAB script that checks for MISRA rule

The script basically checks for two things:

1. MaxIdLength value does not exceed 31 characters
2. Model name does not exceed 19 characters

The model name restriction is due to internal code generation processing that adds certain prefixes to internal data structures based on the model name. This note is described in the MathWorks MISRA Compliance Package, presented earlier in this article.

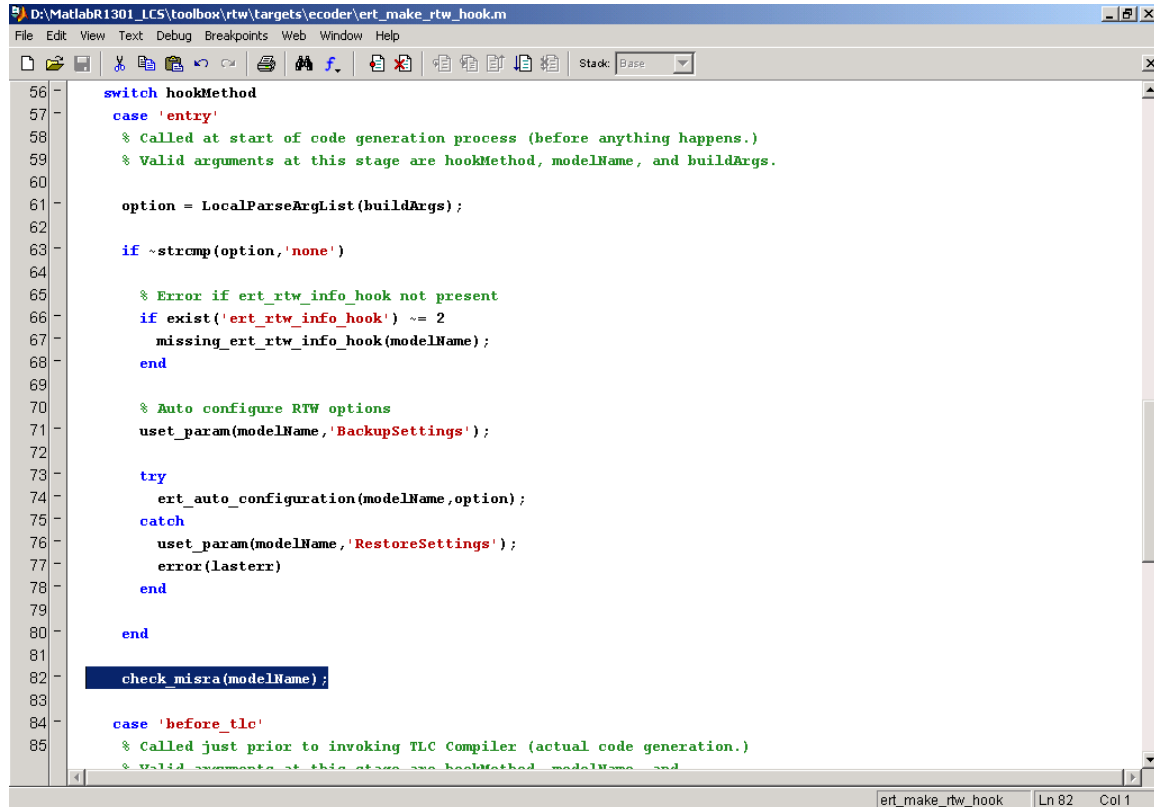
Note that the `uget_param` was used to get `MaxIdLength`. This is a new API that works much like `get_param` except that it supports both Simulink and the Stateflow code-generation parameters. To see the options and settings available for the new `uset_param`/`uget_param`, type the following command:

```
>>uset_param(bdroot,'help')
```

These commands are currently available by installing software as described in the Targeting Tips article in the May 2003 issue of MATLAB Digest but will be available in the next MATLAB release.

Adding the Checker to the ert_make_rt_hook File

The next and final step is to add the MISRA checking script to the ert_make_rt_hook file. As with our earlier change for Splint, this integration process is straightforward and requires a simple change to the file as highlighted in below.



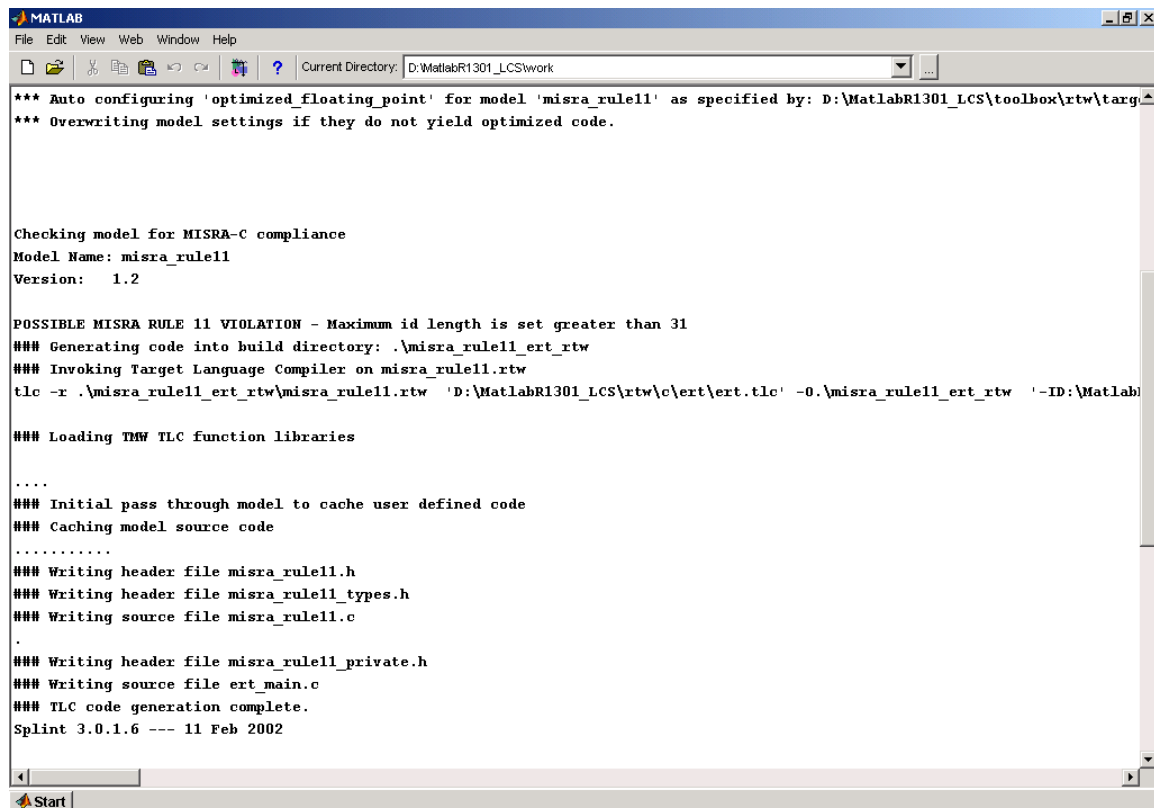
```
56 - switch hookMethod
57 - case 'entry'
58 -     % Called at start of code generation process (before anything happens.)
59 -     % Valid arguments at this stage are hookMethod, modelName, and buildArgs.
60 -
61 -     option = LocalParseArgList(buildArgs);
62 -
63 -     if ~strcmp(option, 'none')
64 -
65 -         % Error if ert_rt_info_hook not present
66 -         if exist('ert_rt_info_hook') ~= 2
67 -             missing_ert_rt_info_hook(modelName);
68 -         end
69 -
70 -         % Auto configure RTW options
71 -         set_param(modelName, 'BackupSettings');
72 -
73 -         try
74 -             ext_auto_configuration(modelName, option);
75 -         catch
76 -             set_param(modelName, 'RestoreSettings');
77 -             error(lasterr);
78 -         end
79 -
80 -     end
81 -
82 -     check_misra(modelName);
83 -
84 - case 'before_tlc'
85 -     % Called just prior to invoking TLC Compiler (actual code generation.)
86 -     % Valid arguments at this stage are hookMethod, modelName, and
```

Figure: Real-Time Workshop Embedded Coder hook file with addition of checker

Notice that the MISRA checker is invoked after the auto configuration steps are performed. This ensures that the checker accesses the final settings that are actually used during code generation.

Building the Model

The model will now be rebuilt and checked using the model-based checker just described. The output from the build process is displayed to the MATLAB command window as shown below. Note that it did detect that the MaxIdlength was set to be greater than 31 (recall it was last set to 100). This script yields a warning but does not “error out”. (This is something that developers could change if desired.). The model’s name length is less than 20 characters so no violation was reported.



```
*** Auto configuring 'optimized_floating_point' for model 'misra_rule11' as specified by: D:\MatlabR1301_LCS\toolbox\rtw\targ
*** Overwriting model settings if they do not yield optimized code.

Checking model for MISRA-C compliance
Model Name: misra_rule11
Version: 1.2

POSSIBLE MISRA RULE 11 VIOLATION - Maximum id length is set greater than 31
### Generating code into build directory: .\misra_rule11_ert_rtw
### Invoking Target Language Compiler on misra_rule11.rtw
tlc -r .\misra_rule11_ert_rtw\misra_rule11.rtw 'D:\MatlabR1301_LCS\rtw\c\ert\ert.tlc' -0.\misra_rule11_ert_rtw -ID:\Matlab
### Loading TMW TLC function libraries
....
### Initial pass through model to cache user defined code
### Caching model source code
.....
### Writing header file misra_rule11.h
### Writing header file misra_rule11_types.h
### Writing source file misra_rule11.c
.
### Writing header file misra_rule11_private.h
### Writing source file ert_main.c
### TLC code generation complete.
Splint 3.0.1.6 --- 11 Feb 2002
```

Figure: Generating code with Checker – Reports MaxIdLength Set above 31 characters

Conclusions

This article demonstrates that it is possible to do a variety of checks at both the model and code levels using MathWorks open interfaces and APIs. The example shown was based on a popular C subset MISRA-C. The benefits of having complimentary automated checks early and late in the code generation process were noted. Finally, the ert_make_rtw_hook file was shown to be a useful integration solution for code checkers such as Splint.

The MISRA Compliance package and new optimized Real-Time Workshop Embedded Coder targets can be requested by e-mail to [Tom Erkkinen \(terkkinen@mathworks.com\)](mailto:terkkinen@mathworks.com).

For more information, including the model files and scripts used in this article [click here](#) or visit us at http://www.mathworks.com/programs/digest_july03/index.shtml.