CEDES

Cost Efficient Dependable Electronic Systems

A Comparison of Three Code Generators for Models Created in Simulink

Author	Robert Hammarström och Josef Nilsson

Document Id	017
Date	14 September 2006
Availability	Public
Status	Final

CHALMERS

A Comparison of Three Code Generators for Models Created in Simulink

ROBERT HAMMARSTRÖM

JOSEF NILSSON

Master's Thesis

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Division of Computer Engineering

Göteborg 2006

CHALMERS

A Comparision of Three Code Generators for Models Created in Simulink

ROBERT HAMMARSTRÖM

JOSEF NILSSON

Master's Thesis

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Division of Computer Engineering

Göteborg 2006

Abstract

As a part of the CEDES project, this report is involved in the work of developing cost efficient dependable electronic systems. The purpose of the master's thesis is to evaluate and compare the three tools Real-Time Workshop Embedded Coder, TargetLink and SCADE Drive. The comparison is based on functionality, compliance to relevant standards, integration with other software and hardware as well as the quality of the generated code. The results are focused on the differences between the tools. TargetLink is integrated in Simulink and is found to have the best user friendliness and graphical interface. SCADE uses a different environment and unlike Real-Time Workshop Embedded Coder and TargetLink it runs without MATLAB and Simulink.

The implementation of fixed-point arithmetic is easily made in TargetLink. When working with Real-Time Workshop Embedded Coder the implementation is made with a tool provided in Simulink. This tool has some weaknesses compared to TargetLink and SCADE.

The means to verify and check the model is also provided in Simulink when working with Real-Time Workshop Embedded Coder. SCADE have several methods available for verification. These extra tools are not the only features distinguishing SCADE. The code generated with SCADE is certified to the standard IEC 61508 and has proven to qualify to the DO-178B standard.

Keywords:
CEDES, Automatic code generation, Simulink, TargetLink, SCADE, Model based

Sammanfattning

Detta examensarbete har utförts som en del i projektet CEDES för utveckling av kostnadseffektiva metoder för felhantering och feltolerans för elektroniksystem i fordon. Syftet med arbetet är att utvärdera och jämföra de tre verktygen Real-Time Workshop Embedded Coder, TargetLink och SCADE Drive. Jämförelsen är baserad på funktionalitet, överensstämmelse med standarder för säkerhetskritiska system, integration med annan mjuk- och hårdvara samt kvalitet på genererad kod. Resultaten är sedan baserade på skillnader mellan de olika verktygen. TargetLink är integrerat i Simulink och har det mest användarvänliga grafiska gränssnittet. SCADE använder sig av en egen utvecklingsmiljö och är i motsatt till Real-Time Workshop Embedded Coder och TargetLink helt oberoende av MATLAB och Simulink.

Omskalning av en modell för fasttalsrepresentation görs smidigt i TargetLink och SCADE medan Real-Time Workshop Embedded Coder använder sig av ett verktyg som finns i Simulink. Detta verktyg innehåller några svagheter om man jämför med TargetLink och SCADE.

Alla tre verktyg tillhandahåller flera möjligheter att verifiera och validera modeller. SCADE är det verktyg som levererar de bästa lösningarna i detta syfte. Som en del i detta är kodgeneratorn certifierad för standarden IEC 61508 och kan ingå som ett led i att utveckla produkter certifierade till DO-178B.

Preface

This Master's project has been carried out at SP, Swedish National Testing and Research Institute in Gothenburg. It is part of a bigger project called CEDES, Cost Efficient Dependable Electronic Systems.

We wish to thank our supervisors at SP, Håkan Edler and Jonny Vinter for their invaluable help during this project. The technical support and friendly attitude from Jonas Cornelsen at Fengco Real Time Control AB, Roger Aarenstrup and Hossein Mousavi at The MathWorks AB and Paul Raistrick at Esterel Technologies Inc. has also been very helpful.

1 Introduction

1.1 Background

The Swedish Parliament stated a safety goal in 1997 called the “vision zero” (sv. Nollvisionen) [1]. It aims to minimise severe injuries in car accidents. In order to reach this goal both vehicles and roads have to be made safer. The construction of dependable and secure cars requires new innovative solutions because of tighter time-to-market and budgets in the automotive industry.

The CEDES (Cost Efficient Dependable Electronic Systems) project [2] is a cooperation between academic institutes and automotive industries in Sweden. Its purpose is to develop cost efficient technologies for safety critical electronic components suitable for vehicles. The vision is to create a system where redundancy, in terms of error-correction and error-tolerance is placed in the software. This approach aims to reduce costs by minimising the amount of hardware used in each car. The software is developed for all vehicles opposite the hardware that is individual for each car.

One of the goals of the CEDES project is to apply and utilise automatic code generation from models when producing software. Both the control algorithm and the simulation of the vehicles behaviour is created as models in Simulink. C code is then generated from the models. There are several tools available for automatic code generation. Model based software development has just started to get acceptance in the industry of safety critical systems. The method has already shown good results in cutting down on development costs and to provide a more lucid overview of the system during the development process.

1.2 Purpose

The aim of this master’s thesis is to analyse, compare and evaluate three different tools for automatic code generation from models in Simulink. The tools being compared are Real-Time Workshop Embedded Coder (RTW-EC) 4.3 from MathWorks [3], TargetLink 2.1.5 from dSPACE [4] and SCADE (Safety-Critical Application Development Environment) Drive 5.1 from Esterel Technologies [5]. MATLAB (MATrix LABoratory) 7.1.0.246 (R14) Service Pack 3 with Simulink 6.3 is the tool used for creating and reviewing models.

1.3 Problem description

Automatic code generation from models is a fairly new technology. It is becoming an accepted and applied method when creating software but a number of problems may arise. This thesis is focused on these questions:

- How well does the generated code fulfil the rules of MISRA C?
- How can the generated code be verified to comply with existing standards for safety-critical systems?
- How can models be validated?
- Are there additional tools required to validate models?
- Are only certain microcontrollers supported or is the generated code general?
- How does the program interact with other software like RTOS, interrupt routines, drivers and existing code?
- What is the quality of the generated code in terms of readability, size and other code metrics?
- How is the code optimised and what are the effects of optimisation?
- What are the limitations when handling Simulink models in the different tools?
- What is the support to return the models to Simulink?
- Are data types other than those using integer arithmetic supported?

2 State-of-the-art

2.1 Model-Based Software Engineering

Model-Based Software Engineering (MBSE) can simply be explained as a way to lower the abstraction level and a way to get a better overview of a project. The objectives of MBSE is to shorten product cycle time and improve product quality and product maintainability. A big step has been taken from the time when programs needed to be written in complex machine code until now when advanced but still user-friendly graphical blocks can be used.

John Von Neumann created two important concepts in 1945 that would come to affect the path of computer programming languages [6]. The first concept was known as “shared-program technique" and stated that the computer hardware should be made simpler and instead controlled by more advanced instructions. This should allow the computer to be reprogrammed much faster than was possible before when the entire system had to be rewired for each new program or calculation. The second concept, "conditional control transfer", added to the ideas of small blocks of code that could be jumped to in any order, instead of a list of instructions that needed to be executed in sequential order. "Conditional control transfer" gave rise to the notion of libraries, which are blocks of code that can be reused over and over again.

The first computer language, called Short Code, for electronic devices was invented in 1949, but it required that the programmer translated its statements into 0’s and 1’s by hand. Still, this was a step towards the advanced algorithms that is being used today.
In 1951, the first compiler was invented which started a new era with many different programming languages. The development went against higher levels of computer language which today has resulted in languages that are using graphical blocks instead of text.

The computing and information technology industry is a field renowned for complexity and customers require more complicated functions and enhanced performance from the products than ever. As an effect of this the demands on the software programmers have become more onerous and difficult. As the number of components in a product increases, so does the numbers of interactions, and thus the possibilities for failures and errors increases.

Many of today companies have seen the advantage of working with models in one way or another. MBSE has become an accepted and established way to develop embedded systems. The number of benefits depends to a great extent on the model. To acquire the benefits from MBSE, it has to be used in all the phases of the project, from developing algorithms, to testing and calibration.

What is needed to create a realistic model is a conceptual approach to modelling that captures vital knowledge of the system. The model should contain not only the structural and functional properties but also its behavioural aspects. Another important characteristic that should be held in mind is that the algorithms shall be reusable, meaning that it should be possible to exploit algorithms in another application. Because of the strong connection between the different steps in the development process (considering that algorithms can be reused in other applications), much of the work can be reused, which focuses the development time to the engineering of the model.

The advantages of using MBSE are among others:

- Engineers can create reusable assets that satisfy a wide variety of uses
- Graphical design enables a high level of abstraction and overview
- Changes in existing software can easily be analysed to quickly compose or synthesize new solutions for subsequent products
- Testing becomes a natural part of construction on all levels
- Higher productivity because of less work
- Fewer errors because of fewer sources
- Speeding up the development process

2.2 Code Generation Tools

Instead of producing code by hand a tool can automatically generate it from models. Blocks in the model represent different operations, for example a mathematical or conditional operation. Signals work their way through the blocks like variables change during execution of a program.

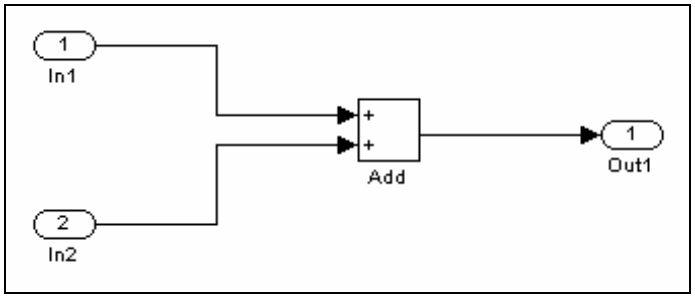


Figure 1 - Addition performed by a Simulink model

```
/* Model output function */
static void add_output()
{
 /* Outport: '<Root>/Out1' incorporates:
  *  Sum: '<Root>/Add'
  *  Inport: '<Root>/In1'
  *  Inport: '<Root>/In2'
  */
  add_Y.Out1 = add_U.In1 + add_U.In2;
}
```

Listing 1 - Addition in a Simulink model (code generated by Real-Time Workshop Embedded Coder)

Automatic code generation from models is especially effective at eliminating syntax errors. Code from a specific block in the model is generated exactly the same every time. As described in *Development of Safety-Critical Software Using Automatic Code Generation* [7], the potential for introducing errors with manual translation is high. By using an automatic code generator, these errors are minimised and the code is kept consistent with the model design. However, using a code generator is no guarantee for getting error free software. The unlimited number of combinations of modelling constructs can lead to errors. Bad modelling will also result in bad code. If the model contains logic errors, these errors will be transferred to the code.

When generating code automatically, standardised functions, comments and documentation are created. Thus the implementation and documents are kept up to date.

The transition from model to code differs between the available tools, but guidelines and standards regulate the outcome. Some standards are very strict and require an extensive certification process.

2.3 Standards

More and more products that are used in our daily life contain some kind of software. Some of these systems are safety-critical, which means that software failures could have catastrophic consequences for the user, especially in cars, aircrafts and other vehicles. To minimize the risk for software failures, different standards and guidelines have been developed. The standards for safety-critical software are among others, safety standard DO-178B (see Appendix V), safety standard IEC 61508 (see Appendix V) and the safe code standard MISRA C, The Motor Industry Software Reliability Association (see Appendix V).

2.4 Developing Safety-Critical Software Using Automatic Code Generation

2.4.1 The V-model

Most standards for development of embedded systems can be modelled by the V-model. This model covers all the steps from defining the system to the complete product and illustrates the stages of the process. Problems detected during the system testing phase, which is carried out late in the development cycle, are often very expensive to fix. Therefore it is desirable to model and analyse the system during early design stages.

Designing a system according to the V-model includes the following phases.

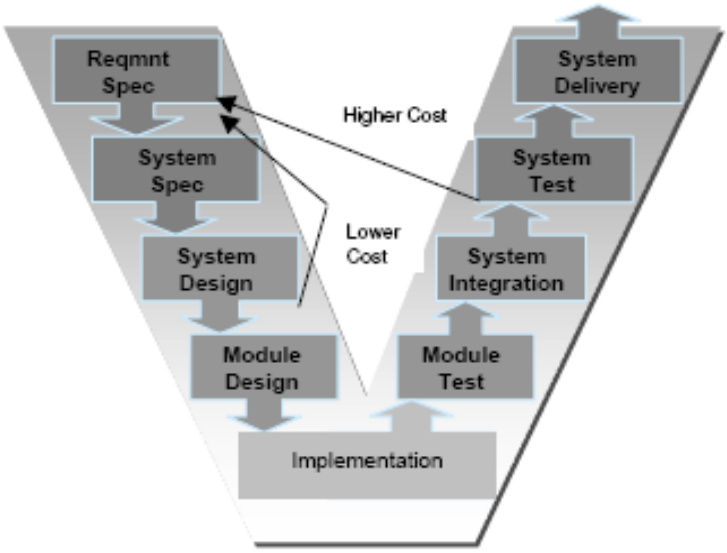


Illustration 1: The V-model [8]

Modelling tools like MathWork's Simulink, dSPACE's TargetLink or Esterel Technologies' SCADE are used to design the models used through the work. These models can then be improved and reused throughout the different phases of the V-model. The model can be, automatically or by hand, translated into code which is implemented on the target hardware. Hardware-in-the-loop mode simulations of the modules and the whole system are made possible. The result is that software developers using modelling tools are provided with automatic code generation and tests in an early stage (the left side of the V-model) of the V-model.

2.4.2 Code generators and standards

All types of safety standards define a set of actions that have to be carried out to achieve a desired safety level. These actions can generally be divided into three categories, selecting development techniques and tools, implementing the system and verifying and validating the system.

When generating code for safety-critical systems, different conditions and restrictions need to be fulfilled. A code generator certified to IEC 61508 or qualifiable for DO-178B shall produce code that can be used in safety-critical applications. The listing below shows some of the features of the SCADE compiler [9] that ensures the high level of safety.

- The tool is deterministic. The input model has a formal definition. Its meaning is completely accurate and is formally defined by the SCADE language.
- The generated code is strictly deterministic. A specific input sequence will always produce the same output sequence. There is no variation as a result of code generation options selected. The behaviour is deterministic based solely on the input model.
- The generated code is safe. It only uses a small, safe subset of the C language. It contains no dynamic memory allocation, no pointer arithmetic and the only loops are bounded loops over delay buffers.
- The generated code is traceable to the input model.

A code generator can be certified to the IEC 61508 standard. This means that the generated code is automatically certified to this standard. The DO-178B standard certifies the final product. The SCADE code generator has been used in the development of products certified to DO-178B and is said to be qualifiable to this standard.

3 Method

When evaluating the three tools for code generation from models created in Simulink several aspects are taken into account. Code generated from all three tools have been compared and tested with the tool QA C[1].

To generate code from the different tools a model from Volvo Technology has been used. The model includes an ABS-system and a simulator of a vehicle. The vehicle part of the model has been used for simulation purposes only and the code being compared has been generated from the ABS-system.

Several aspects of the tools have been tested when evaluating. First is the environment examined and characteristics like user friendliness and interface to settings compared.

The tools for verification and validation included in the tools and some of Simulinks extra toolboxes have been tested and evaluated. Information about the verification and validation tools has mainly been taken from the manuals.

Different ways to customise and optimise the code has been identified and tested. Code is generated with various settings to verify the flexibility and functionality of the available options. Additional models are created to see the effects of the optimisation options available. Both optimised and non-optimised code is generated and compared. Most of the customisation settings are tested, but some require knowledge of the target hardware and have therefore been mentioned but not evaluated.

3.1 Metrics

In an attempt to find concrete values associated with the structure of the code generated from the tools, a set of metrics are computed. The static measurements are calculated by counting different properties in the C code.

The code used in the calculation of the metrics was generated from an ABS-system. Volvo Technology provided the Simulink model which was converted to TargetLink and SCADE.

[1] Programming Research Quality Assurance for C v 4.4.2

The model was fairly simple and not designed to illustrate all aspects of safety-critical real-time systems.

The tool used for the evaluation, QA C, provides a set of metrics, both function-based and file-based. File-based metrics computes a value for each file and function-based present values for each function.

In the report *A comparison between handwritten and automatic generation of C code from SDL using static analysis* [10], a wide range of metrics are computed and used in the comparison between handwritten and automatically generated C code. Among the properties measured are the number of lines of code and the cyclomatic complexity. Both these metrics are discussed in *Metrics that matter* [11], where extra credit is given to the number of code lines metric. The value of this simple metric is presented as one of the best all-around error predictors. A third measured value is the Halstead's Program Level which showed useful when applied on the code evaluated in *Metrics that matter*.

The metric lines of code is quite self explaining, it is the number of lines with executable source code. The value can be calculated for a function (function-based) or a file (file-based). In this report the metric is file-based.

Cyclomatic complexity (VG) or McCabe's cyclomatic complexity [11] is the number of independent paths through the flow graph calculated by:

$$VG = e - n + 2$$

where e is the number of connections between nodes and n is the number of nodes in the flow graph.

The Halstead's Program Difficulty is a metric of the complexity of the code in terms of operators and operands. With a large number of operators and operands the code is considered more difficult to understand.

To complete the list of metrics used in the comparison, the recommendations in *Complexity Analysis of Real Time Software – Using Software Complexity Metrics to Improve the Quality*

of Real Time Software [12] was added. The recommended intervals (Table 1) from this master thesis is applied and used as guidelines. The maximum and minimum values are recommendations made based on the correlation between the metric and errors in code.

Metric	Interval
Cyclomatic Complexity	2-15
Maximum nesting of control structures	1-5
Estimated static path count	4-250
Number of function calls	1-10
Estimated function coupling	1-150
Halstead's program difficulty	-
Number of executable lines	1-70

Table 1 - Limits for software metrics

The maximum nesting of control structures metric measure the maximum control flow nesting. An if-statement with one if and one else condition results in a nesting of two. With an else-if condition added to the statement the value of the metric is increased to three.

An upper bound of the number of paths in the control flow is given by the estimated static path count. The value of this metric is larger or equal to the actual path count which is larger or equal to the cyclomatic complexity.

The number of function calls metric is as the name suggests the total number of calls to functions. It is not the number of unique functions but the total number of calls to functions.

Estimated function coupling (STFCO) is a simplified version of the function coupling metric. The value is calculated from the two metrics, number of function calls (STSUB) and number of function definitions (STFNC).

$$STFCO = \sum(STSUB) - STFNC + 1$$

All tools generate a <'model'.c> file, where 'model' is the name given to the model. This file contains the functions for initialisation, reading input, calculation and updating of output and termination. The metrics presented in this report are based either on the file <'model'.c> (file-based metric) or the functions in that file (function-based metric). Since the three tools generate different sets of functions, the results in this report are presented as file-based, in order to get comparable values. Not all metrics used in the comparison can be calculated as file-based by QA C. Therefore, the function-based metrics are translated to file-based manually.

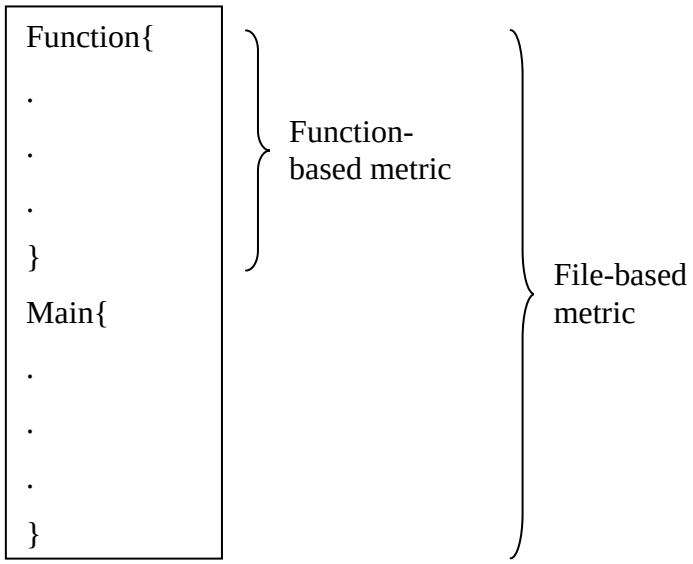


Figure 2 – Illustration of different baisis for metrics

4 Results

4.1 The environment

Both Real-Time Workshop Embedded Coder and TargetLink use the Simulink environment, but to enable code generation with TargetLink, the model needs to be converted first. The conversion between Simulink and TargetLink can be done automatically since many blocks are supported (for a complete list of supported blocks see Appendix VI).

Real-Time Workshop Embedded Coder is a product from MathWorks. It is integrated in the Simulink environment and requires no conversion of models before code can be generated.

The third tool in the comparison is SCADE. It uses a totally different environment for simulation and code generation and therefore a model created in Simulink needs to be translated. Because of the different environment, some settings can be lost in the translation. A translation of a model can be troublesome if it is designed badly or in a non deterministic way. For example in Simulink you do not have to specify the datatype through the model (can be specified as “auto”). In SCADE every datatype needs to be specified to be able to simulate the model.

Both TargetLink and SCADE have support for creation of models without first designing it in Simulink. The scope of this report is not to compare this feature. Instead all models are created in Simulink before they are used in any of the compared tools.

The settings for code generation in TargetLink are all gathered under one dialogue. The dialogue has a simple interface which makes it easy to understand even for an inexperienced user. In Real-Time Workshop Embedded Coder the option pane is more complicated, but provides more options for optimisation of the code (see 4.9).

4.2 Product manual

The manuals provided for the tools are overall good. MathWorks provides a comprehensive search engine on their homepage that contains information about almost everything concerning the Simulink and Real-Time Workshop Embedded Coder environment. Every setting in every pane is explained carefully, making it easy to find information about any

option. The searchable help on the website contains the same help available within MATLAB, but provides some extra features like an FAQ section. This extensive set of documentation, both offline and online, makes technical data easily available to users.

The help in TargetLink is also thorough and explains the different settings well. However, it can sometimes be difficult to find a specific topic, despite the help button found in the dialogue of all TargetLink blocks.

Retrieving technical information in SCADE can sometimes be difficult. The help given in SCADE is focused on solving different problems associated with development and configuration of models. Documentation of technical data is limited to the SCADE Technical Manual which explains many features but is not as detailed as the one provided by MathWorks.

4.3 Model Block support

None of the tools can generate code from all blocks provided by Simulink. To begin with, continuous blocks are not intended for an embedded target. Real-time systems are executed with a sampling rate which is why only discrete blocks are used. Other blocks irrelevant for code generation are for example some of the blocks found in the Simulink Source and Sink libraries. These blocks are used in simulation to log or generate signals and will either be ignored or reporting an error when generating code.

Real-Time Workshop Embedded Coder supports all blocks relevant for code generation and has the fewest limitations to Simulink models. RTW-EC has also support for many of the blocksets that are available for Simulink. TargetLink supports many blocks, but limitations associated with some of them prevent full compatibility with the corresponding Simulink block. An example of a limitation is the state port on the Discrete-Time Integrator block. For TargetLink to support this block the state port needs to be disabled before conversion of the model.

The user-defined and complex functions like the Embedded MATLAB function, the n-dimensional Look-up table and all matrix functions are blocks only supported by Real-Time Workshop Embedded Coder. SCADE and TargetLink both support insertion of legacy-code, where operations not provided by predefined blocks can be inserted. Real-Time Workshop

Embedded Coder not only supports this feature, but since it provides many of the blocks from the Simulink library, like the Embedded MATLAB function, the user is offered additional options when designing systems.

Simulink subsystems have a set of options controlling their behaviour. They can for example be conditionally executed based on If-statements or treated as atomic units[1]. SCADE supports many of the different ports and subsystems available in Simulink. The difference is that the settings become hard-coded when importing a model into SCADE and loops are not allowed, eliminating for and while iterations. This because loops can present a threat to the determinism of the resulting software.

Lack of support for a specific block does not necessarily mean that the desired operation can not be performed, since Simulink provides blocks that combine a set of blocks in one. This feature in Simulink will affect the conversion to TargetLink and SCADE and the user may want to bear this in mind when designing a system.

[1] Subsystems are by default virtual in Simulink. This kind of grouping is only graphical and has no meaning when simulating or generating code. If the subsystem is atomic it is treated as a unit and the structure inside it is not visible to the outside system.

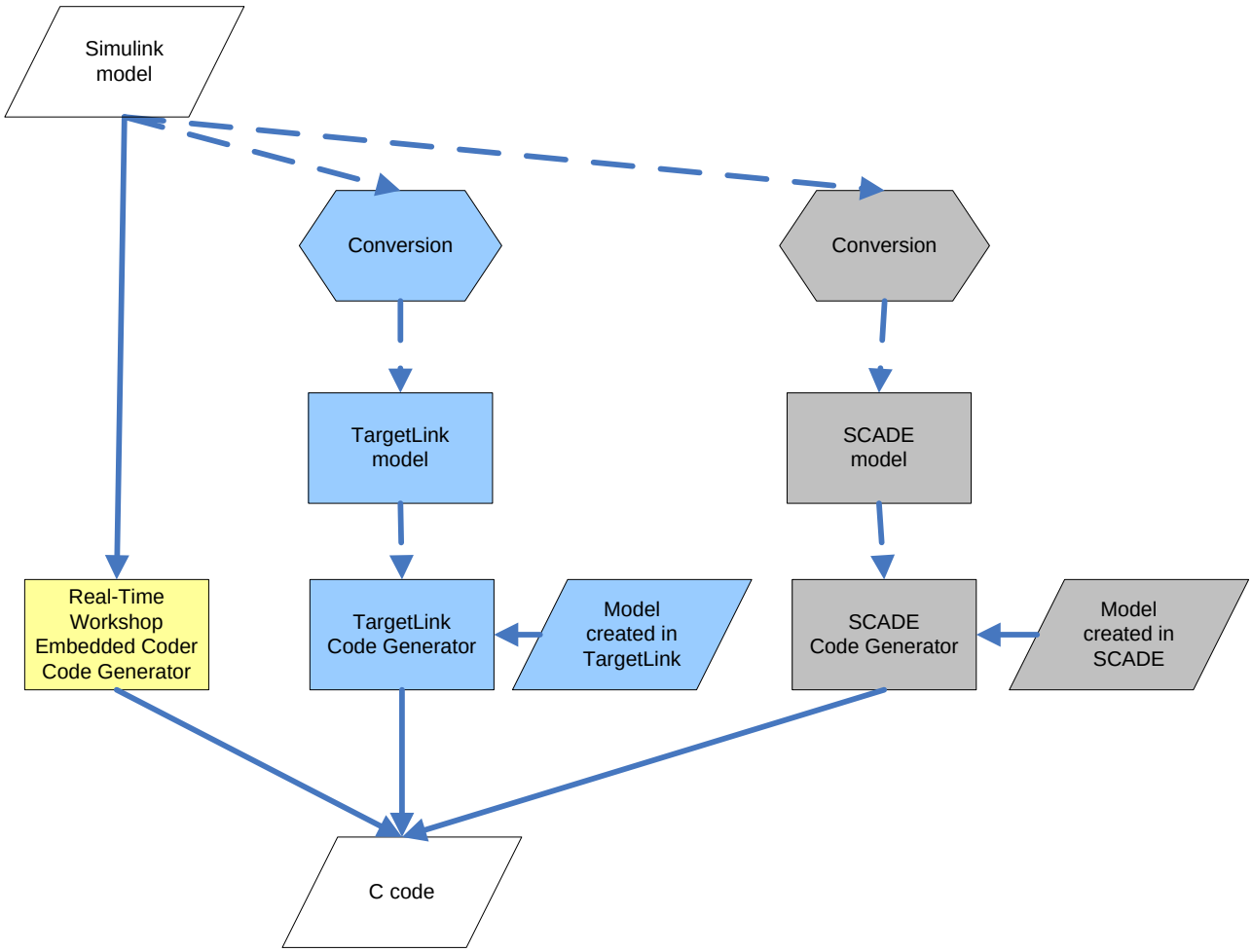


Figure 3 - From model to code

The above figure shows the different paths from model to generated code. This illustrates that Real-Time Workshop Embedded Coder uses Simulink models. The TargetLink and SCADE tools can create their own as well as import models from Simulink.

4.4 Validation and verification

A key issue in model based programming is the accuracy of the model. When designing the system there is often a need to simulate reality which can be complicated. Another source for errors is the special requirements of real-time systems, for example the necessity of correct timing. To meet these requirements, extensive surveys have to be performed on the model.

Simulink, TargetLink and SCADE provide different tools to validate and verify models. All three manufacturers also provide additional tools that can be used to make more extensive tests on the model and the generated code.

SCADE and Simulink have a tool that examines the model and reports blocks that are found unreachable. This is made to prevent existence of unnecessary blocks in the model. TargetLink does not provide this test, but has an additional tool that checks the generated code. The tool goes through every branch in the code to make sure that no unreachable operations have been generated.

All three tools can verify models and check if a signal stays within a specified range. Simulink provides different blocks that assert when a signal leaves a specified range or limit. TargetLink fully supports this set of blocks provided by Simulink. SCADE's tool for verification has the same properties as the blocks in Simulink but also provides some other useful features. More advanced requirements can be specified in SCADE. In Simulink properties based on simulation can only be carried out, SCADE however checks for every possible value mathematically. For example, it is possible to check mathematically if two signals can be true at the same time or not, regardless of the inputs to the model. This is made possible through that a model in SCADE is based of a formal language called LUSTRE.

To validate and verify that the code has fulfilled the requirements, TargetLink provides three simulation modes, model-in-the-loop (MIL), software-in-the-loop (SIL) and processor-in-the-loop (PIL). MIL mode simulates the model and checks controller design and parameterisation as well as behaviour. SIL mode runs the generated code on the host computer and is among other things used to check fixed-point scaling. In Simulink, an S-function (see Appendix I) is created and replacing the original model when performing a SIL mode simulation. PIL mode simulation runs the code on the target processor. Data may be logged in the different modes and compared against each other.

SCADE provides an ability to verify that the functionality of a Simulink model still maintains after conversion. This can be performed by creating an S-function from the imported model. The S-function is then inserted as a block in Simulink allowing interaction with models of the environment.

Simulink provides a useful tool that checks a model for conditions, blocks and settings that can be inappropriate for embedded systems. The tool has two parts, one that checks the model and one that checks the settings for code generation by Real-Time Workshop Embedded

Coder. The tool can therefore be used on a model before converting it to a TargetLink or SCADE model.

4.5 Scheduling and integration with RTOS

No sampling times can be specified in SCADE models. This distinguishes SCADE from both Real-Time Workshop Embedded Coder and TargetLink, but the differences become less remarkable when comparing the generated code. None of the tools control the sampling time of the application, only execution rate as multiples of the sampling time is managed. If different sampling times are specified in TargetLink or Real-Time Workshop Embedded Coder models, code can be generated consisting of a specific task for each sampling frequency. These tasks are then made available to the user, either through a single function or as separate functions. Real-Time Workshop Embedded Coder supports both types whereas TargetLink and SCADE only support separate.

To ensure data integrity and determinism when exchanging data between tasks, Real-Time Workshop Embedded Coder and TargetLink use Rate-Transition blocks. This block ensures that data is transferred safely and not interrupted if for example pre-emption of a task would occur. The need for Rate-Transition blocks is not always obvious to the user, which is why Real-Time Workshop Embedded Coder can insert these automatically. This feature is not available in TargetLink. In SCADE there is no Rate-Transition block since it is considered outside the scope of the tool.

4.6 Fixed-Point

All three tools handle fixed-point arithmetic. There are some differences though. Real-Time Workshop Embedded Coder supports the fixed-point settings available in Simulink. These settings include integers of arbitrary bit size, slope and bias. TargetLink only support integers of 8, 16 or 32 bits. The slope and bias settings can be entered in TargetLink if translated to LSB (least significant bit) and offset. SCADE only support the LSB setting and offset by default. Further configurations, like the slope setting can be made available through scripts. To determine the optimal value of this settings and the choice of data type there are three different procedures available.

If the model is to be used in SCADE, fixed-point scaling should be done after conversion. During translation no fixed-point settings specified in the model are preserved. Since

Simulink is a tool for modelling and not for generating code for embedded systems there is no need for scaling before conversion to SCADE. Fixed-point scaling is target-specific and performed when the modelling of the system is finished. Therefore it is not an issue until after conversion. TargetLink preserves fixed-point settings when converting the model.

The auto-scaling tools available in the three environments provide automatic scaling with varying success. In TargetLink all blocks relevant for scaling have options for it. Ranges can be specified in dialogues and simulation or worst-case propagation values are displayed in the same window. SCADE has an interface similar to TargetLink. Auto-scaling is done with the implement-tool that creates a new model with a wrapper node at top most level of the system. This extra node converts the scaled signals of the in- and outputs to unscaled real values. Automatic scaling can be achieved in Real-Time Workshop Embedded Coder as well, but the tool is not very user friendly and does not support worst-case propagation of ranges.

4.7 Customising the generated code

All tools have different settings to customise the code to a way that suits the developer. One basic property that all three tools have is the possibility to add custom code to a model. The added custom code will then be generated along with the rest of the model.

The three tools also provide the possibility to create separate functions for the different subsystems and nodes or to create one function that covers the whole model. Separate files can also be generated for separate functions. The signals to and from the model or subsystem can be handled as arguments, pointers or as global variables. There are no big differences between the tools in terms of flexibility in creating different functions and creating arguments to these. However, SCADE generates function arguments by default insted of using global variables.

When Real-Time Workshop Embedded Coder generates code from different blocks it uses Target Language Compiler (TLC) files. These files provide information on how every block should be translated to code. To change the code generated from a specific block, these files can be edited. Editing TLC files is a very flexible way to customise the generated code, but it requires knowledge of the TLC.

TargetLink and Real-Time Workshop Embedded Coder have a feature to allocate which memory section variables should be stored in. TargetLink will automatically generate the statements required (pragmas etc). However, this feature is not supported by the ANSI-C standard.

4.8 Optimisation of the generated code

Real-Time Workshop Embedded Coder provides many different options for optimisation. TargetLink and SCADE do not have the same freedom of choice. However, with full optimisation chosen, the differences between the Real-Time Workshop Embedded Coder and TargetLink produced code are small. The SCADE generated code is not as optimised due to the high level of traceability between the generated code and the model.

4.9 Standards

The three tools have different areas of use. SCADE is a tool mainly used in the aerospace and defence business but is now advancing in safety-critical automotive embedded software. TargetLink and RTW-EC are mainly used in the automotive area but also have some applications in the aerospace area.

There are several safety standards for safety critical systems and the three tools generate code that fits these standards more or less. SCADE is however the only tool that generates code with a certified code generator. SCADE's code generator generates code that is certified to the standard IEC 61508. Code generated from SCADE has also proven safe enough for use in products certified to DO-178B. Generating certified code has several benefits which mainly shorten the test and certification process. Code generated from Real-Time Workshop Embedded Coder and TargetLink is also suited for safety related systems but requires more testing. A disadvantage of using a certified code generator is that updates of the generator can not be released as frequently because of the time-consuming certification process.

All three code generators follow most of the rules of MISRA C. TargetLink and SCADE has tested their code generators to the rules of MISRA C (2004) and Real-Time Workshop Embedded Coder to MISRA C (1998) which makes them hard to compare. The rules of MISRA C (1998) contain 93 mandatory and 34 advisory rules and MISRA C (2004) contains 121 mandatory and 20 advisory rules. Some of the rules in MISRA C (1998) have been

removed from the MISRA C (2004). The table below shows how many of the rules the different code generators comply with.

Code generator	Compliance			
	Full	Configurable	Partial	None[1]
TargetLink (MISRA C 2004)	79	24	14	24
SCADE (MISRA C 2004)	99	35	0	7
RTW-EC (MISRA C 1998)	109	13	0	5
1: Some of the rules that are not supported do not depend on the code generators. Some rules of MISRA C are not applicable for automatic code generators.				

Table 2 – Compliance to the rules of MISRA C

Full compliance means that a rule is always met when generating code.

Configurable compliance means that the rule can be fulfilled if certain block or code options are set, it is also possible that restrictions on the model needs to be set.

Partial compliance means that the rule has several statements and that some but not all are fulfilled.

None compliance means that the rule is not supported.

4.10 Metrics

Metric	RTW-EC	TargetLink	SCADE	Interval[1]
Cyclomatic Complexity	11	19	20	2-15
Maximum nesting of control structures	3	2	1	1-5
Estimated static path count	57[2]	34992	1E+05	4-250
Number of function calls	0	0	0	1-10
Estimated function coupling	0	0	0	1-150
Halstead's program difficulty	11,12	18,55	19,99	-
Number of executable lines	45	77	144	1-70

1: Recommended metric value from *"A comparison between handwritten and automatic generation of C code from SDL using static analysis"* [10]

2: See the explanation of estimated static path count.

Table 3 – Different metrics

Cyclomatic complexity is considered an important metric which provides insight into reliability and maintainability. Real-Time Workshop Embedded Coder is the only tool with a value that is considered within the acceptable interval. Some strict programming standards state that the value shall not exceed ten. With this interval, code from none of the tools has acceptable values.

The estimated static path count value of the SCADE generated code is outside the recommended interval. TargetLink also produces code with a value outside the interval. Real-Time Workshop Embedded Coder on the other hand, has an estimated static path count within the interval. One of the reasons for this is that Real-Time Workshop Embedded Coder defines the maximum and saturation operations externally. These operations contain if-statements which would add to the value of the metric if they where included in the evaluation.

The Halstead program difficulty metric does not state a clear limit but a value close to 1 is recommended. None of the tools generate code complying with this criterion, but the value got from the Real-Time Workshop Embedded Coder code is almost half the value of TargetLink's and SCADE's.

The high optimisation level of Real-Time Workshop Embedded Coder is illustrated by the maximum nesting of control structures metric. The value three is the largest of the tools but it is still below the upper limit of the interval. SCADE, which has a metric value of one, has a very simple control structure where if-statements only have one path. Hence, there are no else or if-else paths.

The number of function calls and estimated function coupling are zero simply because the code is not divided into functions by either of tools.

When code is generated with SCADE, the model blocks can be generated to separate functions or combined in one function. The code used to evaluate the metrics was generated as one function. The reason for this is to get a comparable set of files that could be put side by side with the code from Real-Time Workshop Embedded Coder and TargetLink. As a result of this the code became quite complex and exceeded several of the interval limits recommended for real-time systems.

Another reason contributing to the more complex code generated from SCADE is the limited optimisation of the code due to the traceability requirements between generated code and model (see section 4.9).

5 Conclusions

The goal of this master thesis was to compare three different tools for code generation. Many different properties has been tested and evaluated. A summary of some important properties has been collected in table 4, shown below.

	Simulink support	User-friendliness	Customisation options	Standard compliance	Verification tools	Fixed-Point
RTW-EC	High	Medium	High	Low-Medium	Medium	Low-Medium
TargetLink	Medium	High	Medium-High	Low-Medium	Medium	High
SCADE	Low-Medium	Medium	Low	High	High	Medium-High

Table 4 – Grades for different properties included in the tools

RTW-EC supports all relevant blocks in the Simulink block library while TargetLink supports about 50 percent and SCADE 40 percent. However, if this restriction is taken into account when designing the system, this should not be a problem.

TargetLink is the most user-friendly of the three tools. This is mostly because of the simple graphical user interface and the concentration of options relevant for code generation in one place.

SCADE is a tool that generates certified code and therefore does not provide as many ways to customise the code as the other tools. The options for customisation are about the same for TargetLink and RTW-EC.

SCADE generates code certified to the standard IEC 61508. Code generated from RTW-EC and TargetLink is not automatically certified to these standards. Instead this process can be carried out afterwards. Both TargetLink and RTW-EC have code that is part of systems which have been certified according to IEC 61508.

Different tools used to verify and validate a model are included in SCADE and TargetLink. Simulink also provides several methods but those are not included in the RTW-EC toolbox but can be purchased separately. SCADE provides the most powerful tools to validate and verify models.

All three tools have the possibility to auto-scale models to fixed-point arithmetic. SCADE's and TargetLink's tools provide the user friendliest interface and has support for different scaling methods.

6 Reference

[1] NTF. Nollvisionen. (Electronic) Accessible: <http://www.ntf.se/omoss/default7793.asp> (2006-06-08).

[2] Edler, Håkan. CEDES. (Electronic) Accessible: <http://www.cedes.se> (2006-06-08).

[3] MathWorks. (Last modified 2006).The MathWorks (Electronic). Accessible: <http://www.mathworks.se> (2006-06-08)

[4] dSPACE. (Last modified 2006).dSPACE (Electronic) Accessible: <http://www.dspace.com> (2006-06-08)

[5] Esterel Technologies. (Last modified 2006).Esterel-Technologies (Electronic) Accessible: <http://www.esterel-technologies.com> (2006-06-08)

[6] Ferguson, Andrew. (Last modified 2004-11-05). The History of Computer Programming Languages. (Electronic). Accessible: <http://www.princeton.edu/~ferguson/adw/programming_languages.shtml> (2006-06-08)

[7] Michael Beine, Rainer Otterbach and Michael Jungmann (2004) Development of Safety-Critical Software Using Automatic Code Generation. SAE 2004 World Congress & Exhibition; March 2004; Detroit, MI, USA.

[8] Zonghua Gu, Shige Wang, Jeong Chan Kim and Kang G. Shin. (2004-01-02). Integrated Modeling and Analysis of Automotive Embedded Control Systems with Real-Time Scheduling. (Electronic). Accessible: <http://kabru.eecs.umich.edu/aires/paper/gu_sae04.pdf> p.3. (2006-06-08).

[9] Esterel Technologies. (Last modified 2006). RTCA DO-178B. (Electronic). Accessible: <http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation.html> (2006-06-08)

[10] Marcello Becucci, Alessandro Fantechi, Marco Giromini and Emilio Spinicci (2005) A comparison between handwritten and automatic generation of C code from SDL using static analysis. (Electronic). Software – Practise and experience 2005; 35; 1317-1347 Accessible: <http://fmt.isti.cnr.it/WEBPAPER/FULLTEXT.PDF> (2006-02-10)

[11] Tim Menzies, Justin S. Di Stefano, Mike Chapman, Ken McGill (2002) Metrics That Matter. Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE p. 51-57

[12] Krusko, Armin (2004) Complexity Analysis of Real Time Software– Using Software Complexity Metrics to Improve the Quality of Real Time Software. Stockholm: Royal Insitute of Technology, Department of Numerical Analysis and Computer Science

Appendix I - Terminology

RTOS

Real-Time Operating Systems provide an environment for embedded applications in real-time systems. Services included in the RTOS are for example real-time scheduling and synchronization mechanisms.

Scheduling

Real-time applications execute periodically with a rate specified by the sample time.

A real-time system with a single sampling time has a single rate. This type of program needs no extra control over the execution beside the basic sampling rate.

Systems with blocks running at different sampling rates need extra managing. This can be done by dividing the model into tasks based on sampling time. Each subsystem or single block with its own sampling rate is placed in tasks.

If the rates are multiples of each other the application can be controlled using if-branches. The shortest sampling time becomes the base rate. An integer variable is used to remember the sample number. If the base sampling time is 1 second and two subsystems have 2 seconds and 3 seconds as their sampling times, the first would execute on every second and the latter on every third sample.

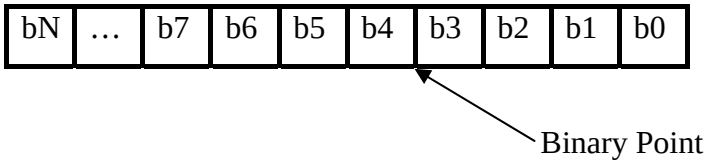
Fixed-point arithmetic

Fixed-point calculations are especially suited for embedded applications, since most of the microcontrollers used do not have a Floating-Point Unit (FPU).

The difference between an ordinary integer and fixed-point data types is the ability to store decimal numbers. Instead of using all bits for the integer value, some are dedicated to fractions of 1.

All fixed-point numbers are evaluated using a ‘slope’ and a ‘bias’. This computation is done through a set of arithmetic operations. To speed up the calculation, the bias can be set to 0 and

the slope is simplified to a power of 2. The result is a binary-point scaling which is done easily by a microcontroller.



Advantages using fixed-point arithmetic are:

a) smaller RAM and ROM consumption,
b) faster execution time and
c) more flexible word size and scaling.

However scaling a whole model to using fixed-point values increases development time and makes the implementation to hardware more complex. It is also easier to get quantization errors due to limited dynamic range.

Auto-scaling

The scaling of signals is a tedious work if done manually. Faults are easily introduced to the system. With auto-scaling this process of calculation and implementation of scaling is handled by algorithms.

To scale a signal properly, a range in which it will operate has to be worked out to eliminate overflow. There are two different approaches to determine the upper and lower limits. If a model is not yet complete or if the model used is not good enough, worst-case scaling can be performed. In the worst-case scenario all known ranges are specified in advance (input signals, constants and so on). The auto-scaling tool then propagates the ranges along the signal lines to blocks that require a worst-case calculation. The alternative approach is to do the range estimation via simulation; the minimum and maximum values are logged during simulation and used as limits to determine the range.

MEX-files

MEX-files provide the possibility to use custom C or FORTRAN routines in MATLAB. Through the external interface (an interface in MATLAB between routines written in other languages and some external communication) the routines can integrate with MATLAB and

be called as if they were M-files (a program written in MATLAB is saved as an M-file) or built-in functions.

S-Function

An S-function is the description of a Simulink block in a computer language. Supported languages are MATLAB, C, C++, Ada and FORTRAN. The S-function is compiled as a MEX-file making it executable in MATLAB.

Simulation modes

There are three different types of simulation modes, Model-In-The-Loop (MIL), Software-In-The-Loop (SIL) and Processor-In-The-Loop (PIL). MIL mode is used for controller design, parametrising and validity checks. It also simulates the model's subsystems to predict whether the current settings will lead to overflows. SIL mode means that the code generated from the model is compiled and executed on the host computer during simulation. Errors concerning the scaling of variables and fixed-point arithmetic effects like quantisation errors are discovered. When simulating in SIL mode the subsystems are disabled and the in- and outputs are redirected to S-function frames that has been generated. PIL mode is used to simulate the generated code on target hardware, this to find errors that are caused by the target compiler or processor. Information like stack usage and execution time can easily be measured in PIL mode. The PIL mode can be used on various target/compiler combinations but requires a license for it.

Appendix II – MathWorks Real-Time Workshop Embedded Coder

The information in the appendix is gathered from MathWorks online manual, the help provided in Matlab and the experiences of the writers of this report.

Supplied by the same company as Simulink, Real-Time Workshop Embedded Coder is well integrated with that modelling environment. There is no need for specific block or translation of blocks before generating code. This feature enables existing models created in Simulink to be used with Real-Time Workshop Embedded Coder without any modifications.

Data types

Simulink models support eight built-in types.

Name	Description
double	Double-precision floating point (64-bit)
single	Single-precision floating point (32-bit)
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer

Table 5 – Data types supported by Simulink

Beside these there are Boolean and Fixed-point data types. The Boolean type is internally represented by uint8 values.

Compared to MATLAB, Simulink support all data types except uint64 and int64.

The default data type of all variables and parameters is double in Simulink. This feature will enable simulation at an early stage of model development. Before code is generated from the model, data types need to be specified to avoid unnecessary memory and processor usage.

Fixed-Point

Support

Fixed-point is supported by Simulink and scaling can be specified using both slope and bias. Both the built-in integer types and integers of arbitrary size up to 128 bits can be used. The arbitrary sized integers can be signed or unsigned and the number of bits allocated for the data type is specified as an integer value.

Name	Description
sfix(TotalBits)	Signed generalized fixed-point data type
ufix(TotalBits)	Unsigned generalized fixed-point data type
sfrac(TotalBits, GuardBits*)	Signed fractional data type
ufrac(TotalBits, GuardBits*)	Unsigned fractional data type
*: GuardBits specifies the number of bits reserved for the integer value	

Table 6 - Fixed-point data type declaration

sfix() and ufix() creates a general fixed-point data type without slope or bias specified. The fractional data types are generated with a fixed binary point. The default placing of this point is to the very left of, or if the data type is signed, immediately to the right of the sign bit. Using fractional data types precludes design with slope and bias.

Auto-Scaling

Simulink can auto-scale a whole model according to logged min and max values. When logging the min and max values, Simulink performs a floating-point simulation. To get the auto-scaling tool to affect the whole model, all blocks need to have their data type mode set to “Specify via dialog”. If this option is not set, the block will not be scaled.

A safety margin can also be set when auto-scaling. This margin is stated in percent and decreases the risk for overflow to occur in the scaled model.

Generated files

When generating code from Real-Time Workshop Embedded Coder the following files are created (depending on the settings used, additional files can be generated):

Model.c:	Contains data definitions and entry functions. Includes the routine model_step that performs the task, model_initialize that initialises the program and model_terminate that causes blocks with terminate functions to execute their termination code.
Model.h	Header file that contains type definitions and aliases of model-specific data structures.
Model_private.h	Contains model-specific macros and data declarations for internal use
Model_data.c	A conditionally generated file containing declarations for the constant I/O blocks and the parameters data structure
Model_types.h	Contains declarations for the real-time model data structure and the parameters data structure
rtw_types.h	Contains type definitions, aliases

Table 7 – Files generated from RTW-EC

Verification and validation

Simulink provides different tools to verify and validate models.

- **Model Advisor**

The Model Advisor checks the entire model or a subsystem for different conditions and settings that can result in inefficient simulation or generation of code that is inefficient or contain code inappropriate for embedded real-time systems. The results are then presented in a report that includes suggestions of settings that can improve the model and the generated

code. The Model Advisor can also be customized by creating an m-file containing defines of custom tasks and checks.

- **Model verification blocks**

Simulink provides several blocks for verification of models. Verification blocks can be set to assert when a signal leaves a specified range or limit. The block can then be set to stop simulation if the signal goes beyond its borders.

- **Requirements Management Interface**

The Requirements Management Interface is a tool used to associate a Simulink model and Stateflow charts with its requirements. The requirements can be typed in a Microsoft Word or a DOORS document. The Requirements Management Interface can then be used to create links between the model in Simulink and the documentation with the requirements. These links can then be used to navigate directly from a requirement to the corresponding block and vice versa. The requirements for each block can also be included as comments in the generated code.

- **Model Coverage**

The Model Coverage tool analyses blocks in the model that directly or indirectly decides the path of the signal. During simulation, the Model Coverage tool saves the behaviour of the different blocks and then reports the extent to which the run exercised potential simulation pathways through each covered block. The tool can be used to find blocks that were not executed during the test run.

- **Simulation modes**

Simulink can perform simulation in both SIL and PIL mode. In SIL mode Simulink creates an S-function wrapper that is an S-function that calls your generated C or C++ code. The S-function can then be integrated in the model to verify that the code has been generated correctly. In PIL mode the code is downloaded to target and simulated. Communications with Simulink during simulation is managed through a serial cable.

Scheduling and integration with RTOS

The implementation of tasks is chosen in the Solver pane of the Configuration Parameters dialogue box. The three modes available are Auto, SingleTasking and MultiTasking.

Auto mode results in SingleTasking if the model uses a single rate and MultiTasking if it uses multi rates.

SingleTasking mode forces the code to use a single task even if the model is a multirate system. The execution of units with different sampling rates is managed by if-statements.

With MultiTasking mode selected, an error is reported if the model only uses a single rate.

Tasking mode	Sampling rate
SingleTasking	Single rate
	Multi rate
MultiTasking	Multi rate

Table 8 – Tasking modes VS sampling rates

The above table shows the available combinations of tasking modes and sampling rates.

With the "**Automatically handle data transfers between tasks"** option checked, rate transition blocks to avoid improper exchange of data between tasks becomes superfluous. Simulink inserts this block hidden in the model to ensure data integrity and deterministic data transfer.

Customising the generated code

Custom code

The Real-Time Workshop Embedded Coder option pane contains a dialogue box for custom C or C++ code to be inserted. Custom code can be inserted in either the source file or the header file, or the initialize function or the terminate function of the generated code. If more flexibility is required for the code placement, Real-Time Workshop Embedded Coder

includes a custom code block library which contains blocks to insert C or C++ code fragments. The code inserted in these blocks will then be added to the code generated from the model but not included when simulating in MIL mode.

Real-Time Workshop also provides a block called Embedded MATLAB function. In the Embedded MATLAB function block, custom MATLAB code can be inserted. The block provides inputs and outputs that will carry the parameters through the code. The block also includes a debugging tool that can be used during simulation of the model.

If C code needs to be inserted, the S-function builder can be used. In the S-function builder custom C code can be inserted. However, the code is translated into an S-function that is used in the model. The S-function builder can also generate a TLC file that is used during code generation of the block.

Target Language Compiler

If a more flexible way to customize the generated code is required, the Target Language Compiler (TLC) files can be modified. The Target Language Compiler is an interpreting language that translates Simulink models into C or C++ code. A TLC file describes how a block in Simulink is going to be translated, so by changing the file it is possible to alter the way code is generated from a particular block. However, changing the files is an advanced option that requires knowledge of the TLC language.

Custom Storage Classes

“Custom storage classes” is a way to adapt signals and parameters in the model to other modules on target. It is therefore possible to control how the generated code stores and represents signals and parameters. A named signal or parameter can have an object with the same name in the MATLAB workspace that describes how the code for the signal/parameter shall look like. With “Custom Storage Classes” it is also possible to assign which memory section variables and constants should be stored in.

Creating functions from subsystems

A subsystem within Simulink can be treated as a virtual subsystem or as an atomic subsystem. If the *treat as atomic subsystem* option is not selected, Simulink treats all blocks in the subsystem as being at the same level as the subsystem. But if the option is enabled Simulink treats the subsystem as a unit when determining the execution order of block methods. The *treat as atomic subsystem* option can also be used to create separate functions from a

subsystem. If desirable, the subsystem is generated in a separate file as an ordinary function or as a reusable function. The name of the function and file can then be chosen as wanted.

Signal properties

Variables (signals) in the code generated by Real-Time Workshop Embedded Coder are stored by default as local variables. If a signal needs to be reached by external written code or if it is desirable to have it declared as a pointer the storage class can be changed. The predefined storage classes available are *Auto, ExportedGlobal, ImportedExtern,* or *ImportedExternPointer*. Auto is the default storage class for signals that do not need to be interfaced to external code. *ExportedGlobal* means that the signal is stored in a global variable. *ImportedExtern* declares the signal as an extern variable and *ImportedExternPointer* declares the signal as an extern pointer.

When the optimisation option “inline parameters” is used, the numerical values of model parameters are used in the generated code, instead of their symbolic names. This means that the parameter has been transformed into a constant and is no longer tunable and therefore not visible to externally written code. However if “inline parameters” is used it is still possible to change the signal property of a specific signal and prohibiting inlining.

Optimisation options

There are several options to customize and optimise the code in Real-Time Workshop Embedded Coder. The following table shows a number of important choices that are available.

	Description	Example
Inline parameters	When this option is checked parameters within the model are inlined, making them nontunable in simulation and inserted as constants in the generated code. The ‘Configuration’-button becomes active when checking *Inline parameters* allowing the user remove parameters from inlining.	See "Example: Inline parameters"

Block reduction	If possible blocks are integrated with each other to create more efficient code. This option affects three types of block reduction: Accumulator Folding, Removal of Redundant Type Conversions, and Dead Code Elimination.	Accumulator folding: Simulink recognizes certain constructs such as accumulators, and reduces them to a single block. Removal of Redundant Type Conversions: Unnecessary type conversion blocks are removed. Dead code elimination: Blocks and signals in an unused path are removed from the generated code.
Enable local block outputs	When this option is selected, block signals are declared locally in functions instead of being declared globally.	
Eliminate superfluous temporary variables (Expression Folding)	Minimizes the computation of intermediate results between blocks. It also collapses block computations into single expressions, instead of generating separate code for each block in the model.	See "Example: Eliminate Superfluous Temporary Variables (Expression Folding)"
Inline Invariant Signals	If a signal is invariant, Real-Time Workshop will precompute and inline it in the generated code.	An invariant signal is a block output signal that does not change during Simulink simulation

Loop Unrolling Threshold	Determines when a vector signal should be included in a for-loop or not.	If the Loop Unrolling Threshold value is set to 5 then the signal have to be wider than 5 to be included in a for-loop. If the signal has fewer elements, separate statement for each element is generated.
Remove code from floating-point to integer conversions that wraps out-of-range values	Removes code that handles out-of-range values. The generated code will still work when the values are within range.	
Generate reusable code	When this option is enabled, data structures are generated as arguments in the model functions. The arguments can be generated as individual arguments or as a struct.	

Table 9 - Optimisation options for RTW-EC

Optimisation examples

The following examples will demonstrate some of the effects of the different options in Real-Time Workshop Embedded Coder.

Example: Inline parameters

This option causes Real-Time Workshop Embedded Coder to use the numerical values of model parameters, instead of their symbolic names, in the generated code. If it is not desirable to inline all parameters, optional settings for each variable can be set.

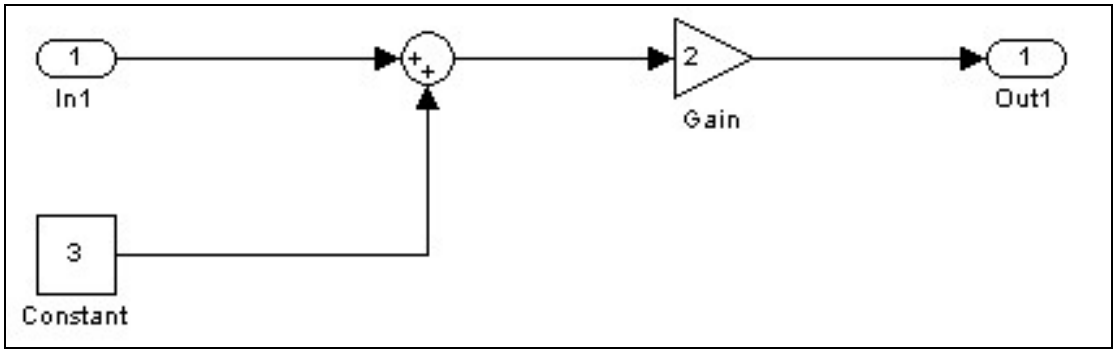


Figure 4 - Simulink model demonstrating the option inline parameters

```
<rtw_example1_data.c>

   16   /* Block parameters (auto storage) */
   17   Parameters_rtw_example1 rtw_example1_P = {
   18     3.0 ,          /* Constant_Value : '<Root>/Constant'
   19                    */
   20     2.0            /* Gain_Gain : '<Root>/Gain'
   21                    */
   22   };
--------------------------------------------------------------------------
<rtw_example1.c>

   26   /* Model step function */
   27   void rtw_example1_step(void)
   28   {
   29
   30     /* Outport: '<Root>/Out1' incorporates:
   31      *  Sum: '<Root>/Sum'
   32      *  Gain: '<Root>/Gain'
   33      *  Constant: '<Root>/Constant'
   34      *  Inport: '<Root>/In1'
   35      */
   36     rtw_example1_Y.Out1 = (rtw_example1_U.In1 +
rtw_example1_P.Constant_Value) *
   37       rtw_example1_P.Gain_Gain;
   38   }
```

Listing 2 - Code generated from model in Figure 4: Inline parameters option disabled

```
<rtw_example1.c>

26   /* Model step function */
27   void rtw_example1_step(void)
28   {
29
30     /* Outport: '<Root>/Out1' incorporates:
31      *  Sum: '<Root>/Sum'
32      *  Gain: '<Root>/Gain'
33      *  Constant: '<Root>/Constant'
34      *  Inport: '<Root>/In1'
35      */
36     rtw_example1_Y.Out1 = (rtw_example1_U.In1 + 3.0) * 2.0;
37   }
```

Listing 3 - Code generated from model in Figure 4: Inline parameters option enabled

Example: Eliminate Superfluous Temporary Variables (Expression Folding)

This option causes Real-Time Workshop Embedded Coder to collapse block computations into single expressions, instead of generating separate code and storage declarations for each block in the model.

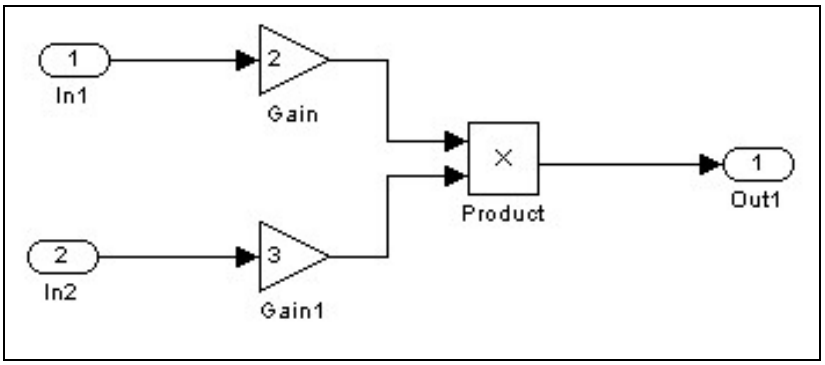


Figure 5 - Simulink model demonstrating the option Eliminate superfluous temporary variables

<rtw_example2.c>

```
26  /* Model step function */
27  void rtw_example2_step(void)
28  {
29
30    /* local block i/o variables*/
31    real_T rtb_Gain1;
32    real_T rtb_Product;
33
34    /* Gain: '<Root>/Gain' incorporates:
35     *  Inport: '<Root>/In1'
36     */
37    rtb_Product = rtw_example2_U.In1 * 2.0;
38
39    /* Gain: '<Root>/Gain1' incorporates:
40     *  Inport: '<Root>/In2'
41     */
42    rtb_Gain1 = rtw_example2_U.In2 * 3.0;
43
44    /* Product: '<Root>/Product' */
45    rtb_Product *= rtb_Gain1;
46
47    /* Outport: '<Root>/Out1' */
48    rtw_example2_Y.Out1 = rtb_Product;
49  }
```

Listing 4 - Code generated from model in Figure 5: Eliminate Superfluous Temporary Variables disabled

```
<rtw_example2.c>
   26   /* Model step function */
   27   void rtw_example2_step(void)
   28   {
   29
   30     /* Outport: '<Root>/Out1' incorporates:
   31      *  Gain: '<Root>/Gain'
   32      *  Gain: '<Root>/Gain1'
   33      *  Product: '<Root>/Product'
   34      *  Inport: '<Root>/In1'
   35      *  Inport: '<Root>/In2'
   36      */
   37     rtw_example2_Y.Out1 = rtw_example2_U.In1 * 2.0 *
(rtw_example2_U.In2 * 3.0);
   38   }
```

Listing 5 - Code generated from model in Figure 5: Eliminate Superfluous Temporary Variables enabled

Appendix III - SCADE Drive

The information in the appendix is gathered from the SCADE manuals, the help provided in SCADE and the experiences of the writers of this report.

SCADE Drive is a Model-Based design environment developed by Esterel Technologies. The tool is specially suited for safety-critical embedded software and includes a certified code generator.

SCADE is a design environment independent on any other software, providing its own blocks and tools for model design and testing. If the model is first developed in Simulink, a conversion is necessary. This feature provides both advantages and disadvantages. One drawback is the translation from the Simulink to the SCADE format which can involve problems. Options and blocks that are not supported in both environments need to be eliminated before conversion and will include extra work. On the other hand, by creating a new model based on the one designed in Simulink, unwanted features of the model can be removed. Simulink is a generic tool suitable for many different types of appplications, not only embedded systems. When generating code for embedded systems, especially safety-critical applications, some options might compromise the safety aspects. For example, blocks that do not meet the strict requirements are not translated.

Data types

Data types supported by SCADE are:

Name	Description
int	Integer
real	Floating point
bool	Boolean
char	Character
string	String (not for code generation)

Table 10 – Data types supported by SCADE

String type is not for code generation. To be able to simulate or generate code, the string has to be converted to an array of characters with a limited size.

Signed and unsigned integers of size 8, 16 and 32 are supported and used when implementing fixed-point arithmetic. 64 bit integers are not included in the standard integer library but can be defined.

Simulink Gateway

SCADE provides a tool to import Simulink models and Stateflow charts which is called the Simulink Gateway. The Simulink gateway can also make automatic updates if the model is being modified in Simulink.

The Simulink Gateway maps the data types when translating from Simulink to SCADE.

Simulink Types	SCADE Types
uint8, int8, uint16, int16, uint32, int32	Int
double, single	Real
boolean	Bool

Table 11 - The translation between data types in Simulink to SCADE

The first step in the translation process is the calculation of arity types within the model. This operation is performed by an algorithm that propagates the type through the model in the same direction as the dataflow. To prevent arity type errors when translating using Simulink Gateway it is recommended that dimensions of in- and outports is specified in the Simulink model.

S-function

When generating an S-function for use in Simulink, a wrapper is created. Blocks for data type conversion are used if necessary to match the Simulink types.

SCADE Types	Simulink Types
Int	Int
Real	Double
Bool	Boolean

Table 12 - The translation between the data types in SCADE to Simulink

Translation of String or Character will report an error since there is no equivalent in Simulink.

Fixed-point

Support

Settings for fixed-point arithmetic can be entered in three ways; Type/Range, Type/LSB or Range/Precision. One of the first two options is used if a specific type is preferred. When Type/Range is utilized, a data type is selected and a range entered. The optimal fixed-point values are then calculated using these settings. If both type and binary point is predefined, these setting are entered in Type/LSB. Optimal fixed-point values can be evaluated using Range/Precision. The minimal integer type is selected automatically and a binary point calculated to match entered values.

Supported integer data types are: int8, int16, int32, uint8, uint16 and uint32.

Auto-scaling

Automatic scaling of signals can be performed through use of simulation ranges or worst-case propagation of ranges. After specifying a root node, a default integer size and a binary point, the implementer is ready to scale the model according to ranges.

If the scaling tool fails or the user wishes to do adjustments to the scaling, values are easily entered in the fixed-point windows of the selected block.

Generated files

When generating code from SCADE, the following files are created:

`nodename.c`	Contains C functions produced from the LUSTRE description. If the **-split** option is used; this file is generated for each nodename unexpanded node.
`nodename.h`	Contains mandatory C declarations. If the **-split** option is used, this file is generated for each nodename unexpanded node.
`Scade_types.h`	Contains the definitions of the types created by the user in the SCADE model.
`nodename_main.h`	If the **-split** option is used, this file is generated and contains scalar constants if **-opt_const** is used.
`nodename_types.c`	Generated when expansion is carried out by functional calls - or mixed functional and inline calls -and when deferred type variables are declared in the model. Contains C functions for conversion of deferred types.
`nodename_types.h`	Generated when expansion is carried out by functional calls - or mixed functional and inline calls -and when deferred type variables are declared in the model. The `nodename_typ.h` file contains macro-instructions used to translate deferred type conversion functions. These macro-instructions use the functions defined in `nodename_typ.c`.
`definitions.h`	Contains macro-instructions declarations for memory copies (`_copy_mem`) and comparisons (`_comp_mem`) and assertions processing (`_assert`).
`Macro_default.h`	Contains default macro-instruction declarations for functions (and for the predefined LUSTRE fby operator if **-macropredef** is set). For example, for a LUSTRE file with the following profile function, `my_function (i1 : int; i2 : bool)` returns `(o1 : real; o2 : int)`.

Table 13 – Files generated from SCADE

Depending on the settings more or less files can be generated.

Verification and validation

- **Model test coverage**

Model test coverage works its way through the model and checks if all element of the SCADE model has been activated. Unreachable blocks in the model are located to eliminate generation of “dead” code.

- **Design Verifier**

The Design Verifier within SCADE can check if the design is according to its requirement, and can be used on the whole model or on one node. An example can be to check if a signal exceeds a specified value or to make sure that two properties not can be true at the same time. The tool can also be used to check whether two models are identical according to the same requirement or not.

- **S-function**

As described earlier in the section about the Simulink Gateway, SCADE provides a tool to verify the model in Simulink.

- **Requirements**

Links can be created between a model in SCADE and a documentation that holds the requirements for the model. The requirements can be typed in a DOORS or Microsoft Word document and can then be used to navigate directly from requirement to corresponding block and back.

Scheduling and integration with RTOS

There is no concept of real time in SCADE. Instead, counting elapsed samples with knowledge of the sampling time is the only way of implementing time into the system. Since no blocks can be assigned a specific sample time, different execution rates in a model is implemented by enabling subsystems on fewer samples than the rest of the system. This technique applicable if the longer sampling time is an integer multiple of the shorter. The implement of conditional execution in SCADE is made with the *CONDAC* operator.

SCADE can generate a task from a node in the model. However, it is not possible to generate multiple tasks automatically. The tasks need to be manually created by generating code from

different nodes. The integration, exchange of data and other issues concerning the environment the application is running on is outside the scope of SCADE. These matters are to be developed by the user.

SCADE can generate wrappers for OSEK and MicroC RTOS's [SCADE RTOS Guidelines]. The files will include OSEK or MicroC and SCADE declarations, an initialisation function and a task function. The integration with other software is handled through globally declared SCADE input and output variables. The variables are buffered to enable safe updating outside the SCADE task.

Customising the generated code

Options during the generation of C code involve naming of variables, expansion of nodes and blocks and the declaration of constants. All these options and a description of them are found in the SCADE Technical Manual.

Split to multiple files

In the code generation settings in SCADE you can choose if the code should be generated to one file or to multiple files. If the "Split to multiple files" option is enabled, SCADE will generate one "node_name.c" and one "node_name.h" file for every node in the model. If the option is disabled the code for every node is generated into one file.

Expansion

In the Expansion pane in the code generation settings selections can be made to combine several nodes to one function. If the option "none" is tagged a function for every node is generated. If the option "selection" is tagged two or more nodes can be combined to one function, used library functions can also be combined here.

Optimisation options

Many of the functions providing optimisation of generated code are not configurable in SCADE. Most of the optimisation functions are therefore always performed and cannot be turned off, however the behaviour of some constants and variables can be configured. Below are some of the functions performed automatically by SCADE.

Dead Code Elimination

All input variables and intermediate signals that is not used to calculate an output is eliminated from the generated code. The only code present in the generated code that is not involved in the calculation of outputs is the code that corresponds to operators with no output, imported operators and global variables.

Variable Elimination

SCADE will optimise variables that are only used once. The local variables that are not optimised are:

> - input and output parameters of a node when they are under an activation condition (if not defined by a type expression)
> - local variables defined as the parameters of the pre operator (they can themselves become memories in case pre operator is optimised)
> - local variables resulting from a case operator (since the assignment is performed in the different cases of a switch)
> - local variables resulting from an if-then-else operator (since they can be allocated in both cases of the if-then-else test)
> - local variables as assertion parameters
> - local variables as probes
> - local variables as the output of a purely functional node
>
> SCADE Technical Manual

Optimisation examples

The model used for the example below is the same as the one used when testing the “inline parameters” option in Real-Time Workshop Embedded Coders.

```
void Node1(_C_Node1 * _C_)
{
    _int _L1_Node1;
    _int _L2_Node1;
    _int _L4_Node1;
    _int _L3_Node1;
    _int _L5_Node1;
/*#code for node Node1 */
    _L1_Node1 = (_C_->_I0_Input1);
    _L2_Node1 = 3;
    _L4_Node1 = (_L1_Node1 * _L2_Node1);
    _L3_Node1 = 2;
    _L5_Node1 = (_L4_Node1 * _L3_Node1);
    (_C_->_O0_Output1) = _L5_Node1;
/*#end code for node Node1 */
}
```

Listing 6 - Code generated from model in Figure 4: No optimisation

```
void Node1(_C_Node1 * _C_)
{
/*#code for node Node1 */
   (_C_->_O0_Output1) = (((_C_->_I0_Input1) * 3) * 2);
/*#end code for node Node1 */
}
```

Listing 7 - Code generated from model in Figure 4: Optimisation on local variables

The model used for the example below is the same as the one used when testing the "Eliminate Superfluous Temporary Variables (Expression Folding)" option of Real-Time Workshop Embedded Coders.

```
void Node2(_C_Node2 * _C_)
{
    _int _L1_Node2;
    _int _L3_Node2;
    _int _L5_Node2;
    _int _L4_Node2;
    _int _L2_Node2;
    _int _L6_Node2;
    _int _L7_Node2;
/*#code for node Node2 */
    _L1_Node2 = (_C_->_I0_Input1);
    _L3_Node2 = 2;
    _L5_Node2 = (_L1_Node2 * _L3_Node2);
    _L4_Node2 = 3;
    _L2_Node2 = (_C_->_I1_Input2);
    _L6_Node2 = (_L4_Node2 * _L2_Node2);
    _L7_Node2 = (_L5_Node2 * _L6_Node2);
    (_C_->_O0_Output1) = _L7_Node2;
/*#end code for node Node2 */
}
```

Listing 8 - Code generated from model in Figure 5: No optimisation

```
void Node2(_C_Node2 * _C_)
{
/*#code for node Node2 */
   (_C_->_O0_Output1) =
                (((_C_->_I0_Input1) * 2) * (3 * (_C_->_I1_Input2)));
/*#end code for node Node2 */
}
```

Listing 9 - Code generated from model in Figure 5: Optimisation on local variables

Appendix IV – TargetLink

The information in the appendix is gathered from the TargetLink manuals, the help provided in TargetLink and the experiences of the writers of this report.

TargetLink is an automatic production code generator created by dSPACE. TargetLink is based on Simulink from MathWorks and uses its environment for modelling the control functionality. The code generator supports many of the blocks used in Simulink, but special TargetLink blocks need to be used to customize and optimise the code.
TargetLinks code generator also supports most of the rules of MISRA C, but it is not DO-178B certified. However the code generated from TargetLink can be certified later on.

"A TargetLink subsystem is a Simulink subsystem prepared for production code generation with TargetLink"

-TargetLink Manual: Production Code Generation Guide

Data types

The built-in data types available in TargetLink are:

TargetLink Types	Description
Int8	Signed 8-bit integer
Uint8	Unsigned 8-bit integer
Int16	Signed 16-bit integer
Uint16	Unsigned 16-bit integer
Int32	Signed 32-bit integer
Uint32	Unsigned 32-bit integer
Float32	32-bit floating-point
Float64	64-bit floating-point
Bool	Boolean

Table 14 – Data types supported by TargetLink

Models designed from scratch in TargetLink or through Simulink with a subsequent translation, will both be simulated using the Simulink simulation engine. When running a model-in-the-loop simulation of a TargetLink model all signals are computed with data type double. Outputs from the subsystem may therefore differ in data type. When generating code, the specified data type of each block is implemented, thus affecting the outcome of a SIL and PIL mode simulation. If the option “Cast option to TargetLink Type” is set, block outputs can be simulated as integer signals.

Simulink to TargetLink

In the translation from Simulink to TargetLink data types are mapped as:

Simulink Types	TargetLink Types
double	Default
single	Float32
boolean	Bool
int8	Int8
uint8	Uint8
int16	Int16
uint16	Uint16
int32	Int32
uint32	Uint32

Table 15 - The translation between data types in Simulink to TargetLink

The Simulink data type double is translated to ‘default’ when converting to TargetLink. Initially the value of default is Int16, but this data type should be chosen to match the targets processor.

The process of mapping data types during a model conversion begins with a compiled mode, followed by a block-by-block mode. In the first stage of the process, some dependencies between blocks in the model are taken into account and data types of different signals may affect each other. Data type selection through inheritance from other blocks can be evaluated

in this mode. In some cases this approach results in undefined types. Therefore the block-by-block mode becomes the next stage in the process. During this step, each block is converted separately, without considering any connected blocks. Use of library blocks require block-by-block mode.

Fixed-point

Support

TargetLink supports both fixed-point and floating-point calculations. Instead of entering slope and bias, TargetLink use the terms LSB and offset. LSB specifies the binary point and the offset is equivalent to bias.

Supported integer types for scaling are: int8, int16, int32, uint8, uint16 and uint32.

Auto-Scaling

TargetLink can auto-scale variables both via worst-case and via use of simulation values. Ranges for use in the worst-case propagation are easily entered in the main dialogue of each block. Limits from previous simulations and range propagations are also displayed in this dialogue.

A safety-margin is entered either in percent or bits. If the margin is entered in bits, the slope will have the number of bits specified added to the calculated value.

Generated files

TargetLink generates a set of files for each TargetLink subsystem that are in the model. By default the following files are generated.

Subsystem.c	Contains the production code for the subsystem.
Subsystem.h	Contains declarations of global variables and functions defined in subsystem.c
Subsystem_udt.h	Contains user-defined types. If no additional types are set, this file will not be created.
tl_defines_<subsystem_ID>.h	Includes TargetLink defined pre-processor macros and log macro. (subsystem_ID: all subsystems have an ID represented by a letter)
tl_basetypes.h and tl_types.h	Contains TargetLink defined types like Int8.

Table 16 – Files generated from TargetLink

Verification and validation

- **Validation checks**

When loading definitions of data types, variable classes, function classes etc., from the Data Dictionary, a validation check is performed. This validation check is performed to check the consistency of the loaded data. The validation check can be performed as “level 3” or “level 4”. Level 3 checks if the data pool in the data dictionary complies with the data model and if the properties of objects have valid values. Level 4 performs the same test as Level 3 but also checks cross-dependencies between objects in the Data Dictionary. When code is generated from the model a validation check of level 4 is performed automatically.

- **Model checking – Invalid blocks, User types**

“Invalid blocks” goes through the whole model searching for blocks not supported by the TargetLink code generator.
“User types” checks if the model contains undefined data types.

- **Model verification blocks**

TargetLink does not have a tool to verify the model, however TargetLink fully supports the verification blocks that are provided by Simulink. Verification blocks can be set to assert when a signal leaves a specified range or limit. The block can then be set to stop simulation if the signal goes beyond its borders.

- **MIL, SIL and PIL mode**

TargetLink uses all three different simulation methods to check the model and the generated code, Model-in-the-loop, Software-in-the-loop and Processor-in-the-loop. TargetLink can also automatically log signals to compare the results from the different simulation modes or to compare the result from before and after auto scaling. The differences between the MIL, SIL and PIL modes can easily be put side by side by plotting the logged signals. This provides an excellent way to see if parameters have been scaled properly and if the production code has been generated correctly.

Scheduling and integration with RTOS

TargetLink is by default configured to generate single tasking, single rate code. Different sampling rates are overrun and the system is configured to use only one rate.

With the option **Enable multirate code generation** checked, different sampling rates are implemented in the code, either as different tasks or by use of if-statements. The implementation can be done manually by inserting TargetLink Task or TargetLink Function blocks in subsystems or left to be done automatically be the code generator. If no task or function blocks are used, TargetLink will group blocks with identical sampling rate in a common task.

Customising the generated code

General block options

In TargetLink every block has a pane that will open when double-clicking on the block. Every pane contains important settings about how a block will appear in the generated code. Some of the settings listed on the block output page are:

Class: If the default class option in TargetLink is selected, the Code Generator automatically selects the most efficient implementation for the output variable and that it is optimized if possible. If another class is desirable, for example if the variable is used as a function argument or as a global variable, this can be changed here. User defined classes can also be created if needed.

Type: Specifies the variables data type. User defined data types can be created if needed.

Name: Lets you specify the name of the variable in the generated production code. Fixed names can be edited or macros can be used.
Address: Here a valid C address for the output variable can be specified. The memory referenced must be reserved and initialized. When generating code, TargetLink will read the input as a pointer to the data type without any further consistency check.

The function block

If it is desirable to make a function out of a subsystem the TargetLink “function block” can be used. The function block contains different options on how the function will appear in the code. Some useful options are.
Step function name: Lets you specify the name of the function in the generated production code. Fixed names can be edited or macros (naming macros) can be used.
Step function class: Specifies a function class for the step function. For example if the function should be generated as a global function or as an extern function.
Make function reusable: Makes the function reusable.
C code file name: Generates the step function to a separate file with the name typed here.

Assigning Variables to a Memory Section

In TargetLink it is possible to allocate which memory section the variables is stored in. This can be done by making the compiler use different memory sections for the variables. These memory sections can allocate memory separately by passing commands to the linker. The name of a memory section is typed in the “SectionName” property, which can be reached by the Data Dictionary. The ANSI C standard does not support allocation of variables to memory sections, so if generic ANSI C code is generated the “SectionName” property is going to be ignored. TargetLink will then automatically generate the statements required (pragmas etc).

Custom code block

TargetLink provides a block called “Custom code block” where custom C code can be inserted. The block has inputs and outputs where data can be sent in- and out of the code. Several choices like declarations of variables, initializing, restart, termination and header code are available. The added code is generated directly into production code without any adjustments; however it needs to be built first. It is also possible to use the generated code for simulation purposes only and exclude it from production code.

When simulating the custom code in MIL-mode, an S-function frame is generated. The frame will then be compiled and linked to an S-function that runs the code.

eXtensible Markup Language

TargetLink uses the eXtensible Markup Language (XML) standard to define the style and layout of the generated code. The XML files can be edited to modify the representation of comments and statement commands. If a more specific and detailed way is needed to customise the appearance of the code, XSL (eXtensible Style sheet Language) files can be changed. However this requires some knowledge about programming in the XSL language.

Optimisation options

The optimisation options within TargetLink are only high-level documented and it is therefore hard to show how some of the choices affect the generated code. TargetLink supports different ways to log variables and signals, but whenever a signal or variable is logged it cannot be optimised. When producing production code, the “clean code” option shall be tagged. This disables all logging activities of macros and variables.

TargetLink can optimise the control flow by eliminating unused if-branches and variables, moving code into conditional branches to avoid superfluous calculations and transform loops with constant conditions into simpler construct. To get TargetLink to perform this optimisation the “Optimisation” option in the Data Dictionary need to be set to “MOVABLE”.

Optimisation examples

The model used for the example below is the same as the one used when testing the “inline parameters” option of Real-Time Workshop Embedded Coders.

```
Void Subsystem(Void)
{
  /* SLLocal: Default storage class for local variables | Width: 16 */
  Int16 Sb1_Gain;
  Int16 Sb1_Sum;

  /* Sum: Subsystem/Sum */
  Sb1_Sum = (Int16) (((UInt16) Sb1_In1_) + 3);

  /* Gain: Subsystem/Gain */
  Sb1_Gain = (Int16) (Sb1_Sum << 1);

  /* Outport: Subsystem/Out1 */
  Sb1_Out1_ = Sb1_Gain;
}
```

Listing 10 - Code generated from model in Figure 4: Optimisation level 0

```
Void Subsystem(Void)
{
  /* Outport: Subsystem/Out1
     # combined # Gain: Subsystem/Gain
     # combined # Sum: Subsystem/Sum */
  Sb1_Out1_ = (Int16) (((Int16) (((UInt16) Sb1_In1_) + 3)) << 1);
}
```

Listing 11 - Code generated from model in Figure 4: Optimisation level 2

The model used for the example below is the same as the one used when testing the "Eliminate Superfluous Temporary Variables (Expression Folding)" option of Real-Time Workshop Embedded Coders.

```
Void Subsystem(Void)
{
  /* SLLocal: Default storage class for local variables | Width: 16 */
  Int16 Sa1_Gain;
  Int16 Sa1_Gain1;
  Int16 Sa1_Product;

  /* Gain: Subsystem/Gain */
  Sa1_Gain = (Int16) (Sa1_InPort << 1);

  /* Gain: Subsystem/Gain1 */
  Sa1_Gain1 = Sa1_InPort1 * 3;

  /* Product: Subsystem/Product */
  Sa1_Product = Sa1_Gain * Sa1_Gain1;

  /* Outport: Subsystem/out. */
  Sa1_OutPort = Sa1_Product;
}
```

Listing 12 - Code generated from model in Figure 5: Optimisation level 0

```
Void Subsystem(Void)
{
  /* Outport: Subsystem/out.
     # combined # Product: Subsystem/Product
     # combined # Gain: Subsystem/Gain
     # combined # Gain: Subsystem/Gain1 */
  Sa1_OutPort = ((Int16) (Sa1_InPort << 1)) * Sa1_InPort1 * 3;
}
```

Listing 13 - Code generated from model in Figure 5: Optimisation level 2

In TargetLink an ‘Optimisation level’ between 0 and 2 can be chosen. This option is only high-level documented and includes many different optimisations. The by dSPACE documented [*p. 498 Advanced practices guide*] effects this option has on the generated code are:

Level	Optimisation
0	No optimisation is performed
1	• Code structures are simplified. • Block outputs are computed only if necessary. • Superfluous block output variables are eliminated, for example, logical and relational blocks are combined in single Boolean expressions. Appropriate code comments are generated. • Superfluous auxiliary variables are eliminated.
2	Completed optimisation, that is, the depth of analysis is increased. For example, the number of iterations is increased.

Table 17 – Optimisation options provided by TargetLink

Menu

The menus provided by TargetLink are uncomplicated and is easy to use. TargetLink offer its own block library and a main dialogue where all options related to TargetLink are located. TargetLink blocks contain additional data for code generation compared to Simulink, such as the scaling information for fixed-point variables, class etc.

Data dictionary

The data dictionary provided by dSPACE is a central data container that holds all data and information for code generation and calibration. All the parameters within the model can be modified directly in the data dictionary which makes it easier if a large number of variables need to be changed. The data dictionary can also be used to share data between engineers working together in a project. Data structures for variable declaration and formulas for fixed-point are some of the features that can be used in the data dictionary. The data stored in the data dictionary can then be used not only by the TargetLink model but also in the entire development process and in dSPACEs other products.

Blockset Stand-Alone

TargetLink has a blockset called “TargetLink Blockset Stand-Alone” which can be used for free and does not require a license. The blockset includes all TargetLink blocks and requires MATLAB/Simulink installed, however none of the functions provided by TargetLink can be used. Models can be modified but features like scaling and code generation is not available.

Limitations

To build models in TargetLink, MathWorks Simulink is used. Blocks that are used in Simulink are easily converted to TargetLink blocks, however not all blocks are supported. TargetLink is designed to generate production code in C and therefore only use models containing discrete blocks. Continuous time blocks are not supported but can be simulated in MIL mode.

Encountered problems

When translating a large Simulink model into TargetLink, some problems may occur if the whole model is converted in one step. To solve this problem the model can be translated in many steps, starting with the subsystem in the deepest layer and then gradual translate one layer at a time. The library link needs to be disabled on all subsystems to make a correct translation.

Appendix V - Standards for safety-critical systems

Safety standard DO-178B/ED-12B

The safety standard DO-178B (in Europe called ED-12B) has been developed by the Radio Technical Commission for Aeronautics (RTCA) for the aviation industry [1]. The standard is primarily concerned with development processes and consists of five certification levels, A, B, C, D or E. Each level describes the consequences of a potential failure of the software: catastrophic, hazardous-severe, major, minor, or no-effect. The different objectives for the different levels are shown in the table bellow [2].

Objectives	Applicability by SW level			
Description	**A**	**B**	**C**	**D**
Test procedures are correct	1	2	2	
Test results are correct and discrepancies explained.	1	2	2	
Test coverage of high-level requirements is achieved	1	2	2	2
Test coverage of low-level requirements is achieved	1	2	2	
Test coverage of software structure (modified condition/decision) is achieved	1			
Test coverage of software structure (decision coverage) is achieved	1	1		
Test coverage of software structure (statement coverage) is achieved	1	1	2	
Test coverage of software structure (data coupling and control coupling) is achieved	1	1	2	
1: The objective should be satisfied with independence. 2: The objective should be satisfied.				

Table 18 - Verification effort table based on the level of software

A product certified to level A would have the largest potential market, but will in turn require more preparations to reach the strict requirements. The DO-178B standard enforces good software development practices, system design processes and describes traceable processes for objectives such as [1]:

- High-level requirements are developed
- Low-level requirements comply with high-level requirements
- Source code complies with low-level requirements
- Source code is traceable to low-level requirements
- Test coverage of high-level and low-level requirements is achieved

Safety standard IEC 61508

Safety standard IEC 61508 covers all safety-related systems that are electronic or programmable. It sets out the requirements for ensuring that systems are designed, implemented, operated and maintained to provide the required safety integrity level (SIL) [3]. According to the risks involved in the system, four safety integrity levels are defined, safety integrity level 1 (SIL1) is the lowest level of safety integrity and safety integrity level 4 (SIL4) is the highest level. The developer need to establish a risk analysis and determine the SIL.

SIL are correlated to the Probability of Failure of Demand (PFD), which is equivalent to the unavailability of a system at the time of a process demand [4].

SIL	ANSI S84.01	PFD	Availability Required	1/PFD
4	NO	10^{-5} to 10^{-4}	> 99.99 %	100,000 to 10,000
3	YES	10^{-4} to 10^{-3}	99.90 to 99.99 %	10,000 to 1,000
2	YES	10^{-3} to 10^{-2}	99.90 to 99.99 %	1,000 to 100
1	YES	10^{-2} to 10^{-1}	99.90 to 99.99 %	100 to 10

Table 19 – Correlation of SIL and PFD

IEC 61508 can be used stand alone or as a basis for other standards and it includes guidelines for requirements, analysis, design, tests, implementation, documentation, validation and product maintenance.

Certifying a code generator to the IEC 61508 standard is both expensive and time consuming; however, it might enable developers to cut down on the volume of verification activities. By not certifying a code generator to the IEC 61508 standard, new updates can be released

frequently. If any modifications are done to a certified code generator, the whole certification process needs to be done all over again.

Safety code standard MISRA C

The commonly used language C was in the beginning considered unsuitable for safety-critical related systems [5]. In 1998 MISRA (Motor Industry Software Reliability Association) produced MISRA C that is a set of rules and guidelines for programming in C, this to make the language more suitable for safety-critical systems. MISRA C contains 121 required and 20 advisory rules.

MISRA C has become the dominant, international coding guidelines for the use of C in critical systems, and is accepted by the IEC 61508 standard.

Referenser

[1] Rose, Greg. (Last modified 2003). Safety-Critical software. (Electronic). Accessible: < http://www.lynuxworks.com/products/whitepapers/safety-critical.php3> (2006-06-08)

[2] Smartworks. Conducting Verification & Validation following DO178B guidelines. (Electronic). Accessible: <http://www.smartworks.us/datasheets/whitepaper3.pdf> p.3 (2006-06-08)

[3] OACG. What is IEC 61508?. (Electronic). Accessible: < http://www.oacg.co.uk/iec61508.html> (2006-06-09)

[4] DYADEM. (Last modified 2006). Introduction to safety integrity levels. (Electronic). Accessible: <http://www.dyadem.com/engineering/risk_management/engineering_services/sil/index.htm> (2006-06-09)

[5] MISRA. (Last modified 2006-03-27). Introduction to MISRA C. (Electronic). Accessible: < http://www.misra-c2.com/> (2006-06-09)

Appendix VI – The ABS-model

To compare code generated from the three tools a model of an ABS-system has been provided. This model has been developed by Volvo Technologies that is a partner within the CEDES project.

The ABS model is still under development and functions such as a fully functional brake pedal, a drive line, steering and other parts are not yet implemented. The brake signal is very simple and can only be set to on or off. When braking, the car will brake from initial speed, without any feedback from the road or vehicle and without clutch, steering wheel and gearbox. The force is equal on all the four wheels, resulting in a straight forward movement of the vehicle. The ABS controller uses a PI controller that has two sets of parameters. The parameters that the PI controller will use depend on the slip.

A simulation of the system has the following scenario. A vehicle is driving with all four wheels straight forward on dry asphalt with friction set to 1. After one second the brake-signal will go from 0 to 1 which means that the brake-pedal is pushed to its maximum. The speed of the wheels and the slip (the relative difference between wheel speed and vehicle speed) is calculated and provided to the PI controller. After three seconds the asphalt will change from dry to wet by setting the friction to 0.7.

Appendix VII - Model Block Support

In the comparison of tools for automatic code generation from Simulink models an important aspect is the support for model blocks. A basis for this analysis has been the blocks support diagram provided in Real-Time Workshop and Real-Time Workshop Embedded Coder. TargetLink compliance to this set of blocks has been evaluated through studies of specifications in the TargetLink user manuals and tests using the conversion tool. The conversion tests are performed to verify automatic translation and to find out if equivalent are blocks are available. All three code generators have imitated support for Simulink models. Some restrictions are due to basic properties of real-time systems, others are related to safety-critical aspects.

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
Additional Math and Discrete: Additional Discrete	Fixed-Point State-Space	N1	N1 Fixed-Point State-Space	-
	Transfer Fcn Direct Form II	N1, N2	-	-
	Transfer Fcn Direct Form II Time Varying	N1, N2	-	-
	Unit Delay Enabled	N1, N2	-	-
	Unit Delay Enabled External IC	N1, N2	-	-
	Unit Delay Enabled Resettable	N1, N2	-	-
	Unit Delay Enabled Resettable External IC	N1, N2	-	-
	Unit Delay External IC	N1, N2	-	-
	Unit Delay Resettable	N1, N2	-	-
	Unit Delay Resettable External IC	N1, N2	N1 Unit Delay Resettable External IC	-
	Unit Delay With Preview Enabled	N1, N2	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Unit Delay With Preview Enabled Resettable	N1, N2	-	-
	Unit Delay With Preview Enabled Resettable External RV	N1, N2	-	-
	Unit Delay With Preview Resettable	N1, N2	-	-
	Unit Delay With Preview Resettable External RV	N1, N2	-	-
Additional Math and Discrete: Increment/Decrement	Decrement Real World	N1	-	-
	Decrement Stored Integer	N1	-	-
	Decrement Time To Zero	X	-	-
	Decrement To Zero	N1	-	-
	Increment Real World	N1	-	-
	Increment Stored Integer	N1	-	-
Continuous	Derivative	N3, N4	-	N1 Derivate
	Integrator	N3, N4	-	-
	State-Space	N3, N4	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Transfer Fcn	N3, N4	-	-
	Transport Delay	N3, N4	-	-
	Variable Time Delay	N3, N4	-	-
	Variable Transport Delay	N3, N4	-	-
	Zero-Pole	N3, N4	-	-
Discontinuities	Backlash	N2	N1 Backlash	X
	Coulomb & Viscous Friction	N1	-	X
	Dead Zone	X	N1 Dead Zone	X
	Dead Zone Dynamic	N1	-	-
	Hit Crossing	N4	-	N1,N2 Hit Crossing
	Quantizer	X	-	X
	Rate Limiter	N5	X	N1,N2 Rate Limiter
	Rate Limiter Dynamic	N1, N5	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Relay	X	X	X
	Saturation	X	X	X
	Saturation Dynamic	N1	-	-
	Wrap To Zero	N1	-	-
Discrete	Difference	N1	-	-
	Discrete Derivative	N2, N6	-	N1 Discrete Derivative
	Discrete Filter	N2	N1, N2 Discrete Filter	N1 Discrete Filter
	Discrete State-Space	N2	N1, N2 Discrete State-Space	-
	Discrete Transfer Fcn	N2	N1 Discrete Transfer Fcn	N1 Discrete Transfer Fcn
	Discrete Zero-Pole	N2	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Discrete-Time Integrator	N2, N6	N1, N2, N3, N4, N5, N6 Discrete-Time Integrator	N1, N2, N3 Discrete-Time Integrator
	First-Order Hold	N4	-	-
	Integer Delay	N2	-	-
	Memory	X	-	-
	Transfer Fcn First Order	N1	-	-
	Transfer Fcn Lead or Lag	N1	-	-
	Transfer Fcn Real Zero	N1	-	-
	Unit Delay	N2	X	X
	Weighted Moving Average	X	-	-
	Zero-Order Hold	X	X	N1 Zero-Order Hold
Logic and Bit Operations	Bit Clear	X	N1 Bit Clear	-
	Bit Set	X	N1 Bit Set	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Bitwise Operator	X	X	N1, N2 Bitwise Operator
	Combinatorial Logic	X	-	N1 Combinatorial Logic
	Compare to Constant	X	-	-
	Compare to Zero	X	-	-
	Detect Change	N2	-	-
	Detect Decrease	N2	-	-
	Detect Fall Negative	N2	-	-
	Detect Fall Nonpositive	N2	-	-
	Detect Increase	N2	-	-
	Detect Rise Nonnegative	N2	-	-
	Detect Rise Positive	N2	-	-
	Extract Bits	X	-	-
	Interval Test	X	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Interval Test Dynamic	X	-	-
	Logical Operator	X	X	N1 Logical Operator
	Relational Operator	X	X	X
	Shift Arithmetic	X	N1 Shift Arithmetic	-
Lookup Tables	Cosine	N1	-	-
	Direct Lookup Table (n-D)	N2	X	-
	Interpolation (n-D)	X	X	-
	Lookup Table	X	N1 Lookup Table	N1 Lookup Table
	Lookup Table (2–D)	X	N1 Lookup Table (2–D)	N1 Lookup Table (2–D)
	Lookup Table (n-D)	X	-	-
	Lookup Table Dynamic	X	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	PreLookup Index Search	X	N1 PreLookup Index Search	-
	Sine	N1	-	-
Math Operations	Abs	X	X	X
	Algebraic Constraint	-	-	-
	Assignment	N2	N1 Assignment	-
	Bias	X	-	-
	Complex to Magnitude-Angle	X	-	-
	Complex to Real-Imag	X	-	-
	Dot Product	X	-	-
	Gain	X	N1 Gain	X
	Magnitude-Angle to Complex	X	-	-
	Math Function (10^u)	X	X	X
	Math Function (conj)	X	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Math Function (exp)	X	X	X
	Math Function (hermitian)	X	-	-
	Math Function (hypot)	X	X	X
	Math Function (log)	X	X	X
	Math Function (log10)	X	X	X
	Math Function (magnitude^2)	X	-	X
	Math Function (mod)	X	X	X
	Math Function (pow)	X	X	X
	Math Function (reciprocal)	X	X	X
	Math Function (rem)	X	X	X
	Math Function (square)	X	X	X
	Math Function (sqrt)	X	X	X
	Math Function (transpose)	X	-	-
	MinMax	X	X	N1 MinMax

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	MinMax Running Resettable	X	-	-
	Polynomial	X	-	-
	Product	N2	N1, N2, N3 Product	N1, N2 Product
	Real-Imag to Complex	X	-	-
	Reshape	X	-	-
	Rounding Function	X	N1 Rounding Function	X
	Sign	X	X	X
	Sine Wave Function	X	-	-
	Slider Gain	X	-	X
	Sum	X	X	N1, N2 Sum
	Trigonometric Function	N7	N1, N2, N3 Trigonometric Function	X
	Unary Minus	X	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Weighted Sample Time Math	X	-	-
Model Verification	Assertion	X	X	-
	Check Discrete Gradient	X	X	-
	Check Dynamic Gap	X	X	-
	Check Dynamic Lower Bound	X	X	-
	Check Dynamic Range	X	X	-
	Check Dynamic Upper Bound	X	X	-
	Check Input Resolution	X	X	-
	Check Static Gap	X	X	-
	Check Static Lower Bound	X	X	-
	Check Static Range	X	X	-
	Check Static Upper Bound	X	X	-
Ports & Subsystems	Atomic Subsystem	X	N1 Atomic Subsystem	N1 Atomic Subsystem

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Code Reuse Subsystem	X	N1 Code Reuse Subsystem	-
	Configurable Subsystem	X	X	-
	Enabled Subsystem	X	N1 enabled subsystem	N1, N2 Enabled Subsystem
	Enabled and Triggered Subsystem	X	-	N1 Enabled and Triggered Subsystem
	For Iterator Subsystem	X	X	-
	Function-Call Generator	X	X	-
	Function-Call Subsystem	X	N1, N2 Function-Call subsystem	-
	If	X	N1 If	N1 If
	If Action Subsystem	X	N1, N2 If Action Subsystem	N1, N2 If Action Subsystem
	Model	X	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Subsystem	X	X	N1 Subsystem
	Switch Case	X	N1 Switch Case	N1 Switch Case
	Switch Case Action Subsystem	X	N1, N2 Switch Case Action Subsystem	N1, N2 Switch Case Action Subsystem
	Triggered Subsystem	X	N1 Triggered subsystem	N1, N2 Triggered Subsystem
	While Iterator Subsystem	X	N1 While Iterator Subsystem	-
Signal Attributes	Data Type Conversion	X	X	X
	Data Type Conversion Inherited	X	-	-
	Data Type Duplicate	X	-	-
	Data Type Propogation	X	-	-
	Data Type Scaling Strip	X	-	-
	IC	N4	-	X

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Probe	X	-	-
	Rate Transition	N2, N5	X	-
	Signal Conversion	X	-	-
	Signal Specification	X	-	-
	Weighted Sample Time	X	-	-
	Width	X	-	N1 Width
Signal Routing	Bus Assignment	X	-	-
	Bus Creator	X	N1 Bus Creator	N1 Bus Creator
	Bus Selector	X	X	N1 Bus Selector
	Data Store Memory	X	N1, N2 Data Store Memory	-
	Data Store Read	X	X	-
	Data Store Write	X	X	-
	Demux	X	N1 Demux	N1, N2 Demux

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Environment Controller	X	-	-
	From	X	N1 From	N1 From
	Goto	X	N1 Goto	N1 Goto
	Goto Tag Visibility	X	X	-
	Index Vector	X	N1 Index Vector	-
	Manual Switch	N4	-	-
	Merge	X	N1, N2 Merge	N1, N2 Merge
	Multiport Switch	N2	X	N1 Multiport Switch
	Mux	X	X	N1 Mux
	Selector	X	N1 Selector	N1 Selector
	Switch	N2	X	X
Sinks	Display	N8	N1 Display	-
	Floating Scope	N8	N1 Floating Scope	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Output (Out1)	X	X	N1, N2 Output
	Scope	N8	N1 Scope	-
	Stop Simulation	-	-	-
	Terminator	X	X	X
	To File	N4	N1 To File	-
	To Workspace	N8	N1 To Workspace	-
	XY Graph	N8	N1 XY Graph	-
Sources	Band-Limited White Noise	N5	-	-
	Chirp Signal	N4	-	-
	Clock	N4	-	-
	Constant	X	X	N1 Constant
	Counter Free-Running	N4	-	-
	Counter Limited	N1	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Digital Clock	N4	-	-
	From File	N8	-	-
	From Workspace	N8	-	-
	Ground	X	X	N1 Ground
	Inport (In1)	X	X	N1 Inport
	Pulse Generator	N5, N9	-	-
	Ramp	N4	-	-
	Random Number	X	-	-
	Repeating Sequence	N10	-	-
	Repeating Sequence Interpolated	N1, N5	-	-
	Repeating Sequence Stair	N1	-	-
	Signal Builder	N4	-	-
	Signal Generator	N4	-	-
	Sine Wave	N6, N9	-	-

Sublibrary	Block	Real-Time Workshop Embedded Coder	TargetLink	SCADE Drive
	Step	N4	-	-
	Uniform Random Number	X	-	-
User-Defined	Embedded MATLAB Function	X	**?**	-
	Fcn	X	N1, N2 Fcn	N1 Fcn
	MATLAB Fcn	N11	-	-
	S-Function	N12	N1 S-Function	-
	S-Function Builder	X	N1 S-Function Builder	-

Table 20 – Supported blocks

Real-Time Workshop Embedded coder notes

Symbol	Note
X	Real-Time Workshop support.
N1	Real-Time Workshop does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more optimal code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
N2	Generated code relies on `memcpy` or `memset` (`string.h`) under certain conditions.
N3	Consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support code generation. To start the Model Discretizer, click **Tools** > **Control Design**.
N4	Not recommended for production code.
N5	Cannot use inside a triggered subsystem hierarchy.
N6	Depends on absolute time when used inside a triggered subsystem hierarchy.
N7	The three functions — `asinh`, `acosh`, and `atanh` — are not supported by all compilers. If you use a compiler that does not support these functions, Real-Time Workshop issues a warning message for the block and the generated code fails to link.
N8	Ignored for code generation.
N9	Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
N10	Consider using the Repeating Sequence Stair or Repeating Sequence Interpolated block instead.

Symbol	Note
N11	Consider using the Embedded MATLAB block instead.
N12	Real-Time Workshop does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more optimal code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.

Table 21 – RTW-EC notes

TargetLink notes

Symbol	Note
X	TargetLink support.
N1 Function-Call Subsystem	Only supports ‘inherit’ of the **States when enabling property**.
N2 Function-Call Subsystem	Output block setting **Output when disabled** needs to be set to ‘held’.
N1 Triggered Subsystem	The **Show output port** option is not supported at root level of the TargetLink subsystem.
N1 Enabled Subsystem	The output port of an Enable block at root level of a TargetLink subsystem will always emit a signal equal to 1 in MIL simulation mode. In SIL and PIL mode the signal is set to the correct value.
N1 Fixed-Point State-Space	No special state-space block for fixed-point data. Uses Discrete State-Space Block which handles fixed-point-values.
N1 Backlash	Hardcoded implementation in a *Custom Code* block.
N1 Dead Zone	Hardcoded implementation in a *Custom Code* block.

Symbol	Note
N1 Discrete filter	The data types of input and output must be of the same. Either integers or floating-point.
N2 Discrete filter	The variable **SLFilterFunctionReturn** is not evaluated.
N1 Discrete State-Space	The data types of input and output must be of the same. Either integers or floating-point.
N2 Discrete State-Space	The size of the input, state and output matrices are not implicitly set. These parameters have to be set manually.
N1 Discrete Transfer Fcn	The data types of input and output must be of the same. Either integers or floating-point.
N1 Discrete-Time Integrator	Does not support **Show state port**.
N2 Discrete-Time Integrator	Does not support the **level** value of the **External reset property**. This property will be mapped to **either-edge**.
N3 Discrete-Time Integrator	Does not support the **Accumulation** values of the **Integrator method**.
N4 Discrete-Time Integrator	Does not support **Gain value**.
N5 Discrete-Time Integrator	Does not support vectorised signals.
N6 Discrete-Time Integrator	Model update will not evaluate the limit output flag.

Symbol	Note
N1 Unit Delay Resettable External IC	The block will not be replaced during a conversion, but has a counterpart called **Unit Delay Reset Enabled** in TargetLink.
N1 Bit Clear	The block will not be replaced during a conversion, but has counterparts for Uint8, Uint16 and Uint32 in **TargetLink Sample Blocks-Bitwise Operators** in TargetLink.
N1 Bit Set	The block will not be replaced during a conversion, but has counterparts for Uint8, Uint16 and Uint32 in **TargetLink Sample Blocks-Bitwise Operators** in TargetLink.
N1 Shift Arithmetic	Does not support binary point shifting.
N1 Lookup Table	Does not support floating-point types with equidistant tables.
N1 Lookup Table (2–D)	Does not support floating-point types with equidistant tables.
N1 PreLookup Index Search	Index and fraction vary between TargetLink and Simulink but the output from the Interpolation (n-D) block is always the same as Simulink.

Symbol	Note
N1 Assignment	Only supports identical types and scaled elements for vectors.
N2 Assignment	Simulation behaviour may differ from Simulink if the block is part of a loop, if the output elements are not initialised before each iteration or if not all output elements have been accessed.
N1 Gain	To multiply with a matrix use the Discrete State-space block
N1 Product	Does not support multiplication of two arbitrarily scaled 32-bit integers. Instead use power-of-two scaling.
N2 Product	Does not support more then two inputs.
N3 Product	Does not support the '//' operation. The '/' operation is only supported for signals with a width of one.
N1 Rounding Function	Hardcoded implementation in a *Custom Code* block.
N1, N2, N3 Trigonometric Function	The fixed-point implementation of **atan** function uses the constant 1/(LSB(input)^2). If this value can not be represented correctly by a 32-bir integer an error will be reported during code generation.
N2 Trigonometric Function	Does not support 32-bit integer as input for **atan2** if implementing fixed-point.
N3 Trigonometric Function	Does not support unsigned input or output for **atan2** if implementing fixed-point.

Symbol	Note
N1 Atomic Subsystem	The input to an atomic subsystem can not be complex if it is specified as a function argument. This input setting also restricts the signal to consist of only data, rather then both data and event signals. These limitations apply to bus inputs as well.
N1 Code Reuse Subsystem	To create a code reuse subsystem in TargetLink a **Function** block is placed at topmost level of that subsystem.
N1 If	See **If Action Subsystem**.
N1 If Action Subsystem	An action port at topmost level of a TargetLink subsystem can not be triggered from outside this system.
N2 If Action Subsystem	An action port outside a TargetLink subsystem can not be triggered from inside the TargetLink system.
N1 Switch Case	See **Switch Case Action Subsystem.**
N1 Switch Case Action Subsystem	An action port at topmost level of a TargetLink subsystem can not be triggered from outside this system.
N2 Switch Case Action Subsystem	An action port outside a TargetLink subsystem can not be triggered from inside the TargetLink system.

Symbol	Note
N1 While Iterator Subsystem	The block can not have an external source for its iteration condition if it resides in the TargetLink root system.
N1 Bus Creator	Does not support the nonvirtual bus option of the Output.
N1 Data Store Memory	The block has to be placed inside the TargetLink subsystem to work together with the Data Store Read and Data Store Write blocks.
N2 Data Store Memory	Does not support the variable to be defined as a function return value, a reference parameter or a value parameter.
N1 Demux	Cannot inherit its signal assignment from its successor.
N1 From	From and Goto connections can not cross boundaries of atomic subsystems. With a TargetLink Function, Task or ISR blocks contained in a subsystem will make it atomic regardless of the option **Treat as atomic subsystem**.
N1 Goto	From and Goto connections can not cross boundaries of atomic subsystems. With a TargetLink Function, Task or ISR blocks contained in a subsystem will make it atomic regardless of the option **Treat as atomic subsystem**.
N1 Index Vector	When converting from Simulink to TargetLink this block becomes a Multiport Switch block with a single input.

Symbol	Note
N1 Merge	Does not support inputs of unequal port widths. Thus the **Allow unequal port widths** checkbox in the Simulink blocks is not supported.
N2 Merge	The simulation sequence is different from Simulink which can cause deviating simulation results. It can be prevented by controlling the sequence using enable signals.
N1 Selector	If **Source of element indices** is set to *External* all inputs need to be of the same data type and scaling.
N1 Display	Supported but irrelevant for production code, thus ignored by the code generator.
N1 Floating Scope	Supported but irrelevant for production code, thus ignored by the code generator.
N1 Scope	Supported but irrelevant for production code, thus ignored by the code generator.
N1 To File	Supported but irrelevant for production code, thus ignored by the code generator.
N1 To Workspace	Supported but irrelevant for production code, thus ignored by the code generator.

Symbol	Note
N1 XY Graph	The XY Graph block is not supported but it can be implemented using the Simulink **Signal & Scope Manager**.
N1 From File	Supported but irrelevant for production code, thus ignored by the code generator.
N1 From Workspace	Supported but irrelevant for production code, thus ignored by the code generator.
N1 Fcn	Does not support overflow checking.
N2 Fcn	Only outputs a scalar. For vector signals us the Math block.
N1 S-Function	S-Functions are implemented using the **custom-code** block which also works similar to the **S-Function Builder** block.
N1 S-Function Builder	The corresponding block is the **custom-code** block. It will not be translated during conversion but serves the same purpose.

Table 22 - Notes

SCADE Drive notes

Symbol	Note
X	SCADE support.
N1 Derivate	The translator calculates a clock that may be different from Simulink. Thus simulation results may be different.
N1 Hit Crossing	**Show Output Port** is always set to “on”.
N2 Hit Crossing	When **Hit crossing direction** is set to *either* the behaviour is an OR between rising and falling. In Simulink this setting serves as “almost equal”.
N1 Rate Limiter	Only supported if **Initial condition** set to “0”.
N2 Rate Limiter	**Sample time mode** should be set to “inherited” to ensure faithful translation.
N1 Discrete Derivative	Only supported if **Initial condition for previous weighted input K*u/Ts** is set to “0.0” and **Gain value** is set to “1.0”.
N1 Discrete Filter	Hard-coded

Symbol	Note
N1 Discrete Transfer Fcn	Hard-coded
N1 Discrete-Time Integrator	Only supported if **Use initial condition as initial and reset value for** is set to “State and Output” and **Gain value** is set to “1.0”.
N2 Discrete-Time Integrator	Does not support **Show state port** or **Limit output**.
N3 Discrete-Time Integrator	Does not support **Lower saturation limit** set to “Inf” and **Upper saturation limit** set to “Inf” or “-Inf”.
N1 Lookup Table	Hard-coded
N1 Lookup Table (2–D)	Hard-coded
N1 Zero-Order Hold	Conditional activation with the SCADE block **condact**.
N1 Bitwise Operator	Only supports **Operator** “AND”, “OR”, “XOR” and “NOT”.
N2 Bitwise Operator	Only supported if **Use bit mask …** checked and **Treat mask as** is set to “Stored Integer”
N1 Combinatorial Logic	Hard-coded

Symbol	Note
N1 Logical Operator	Hard-coded except if **Operator** is set “NOT”.
N1 MinMax	Hard-coded if more than one input. If only one input, the block is translated to an empty block.
N1 Product	Hard-coded if **Number of inputs** is set to other than “/”, “*” or “1”. Inputs=”/” outputs the inverse of the input. “*” or “1” outputs the input.
N2 Product	Does not support **Saturate on integer overflow**.
N1 Sum	Hard-coded if **Number of inputs** is set to other than “-”, “+” or “1”. Inputs=”-” outputs the negative of the input. “+” or “1” outputs the input.
N2 Sum	Does not support **Saturate on integer overflow**.
N1 Atomic Subsystem	Hard-coded
N1 If Action Subsystem	Only supported if **States when execution is resumed** is set to “held”.
N2 If Action Subsystem	Hard-coded

Symbol	Note
N1 Enabled Subsystem	Only supported if **States when execution is resumed** is set to “held”.
N2 Enabled Subsystem	Hard-coded
N1 Enabled and Triggered Subsystem	See Enabled Subsystem and Triggered Subsystem
N1 If	Hard-coded
N1 Inport	Hard-coded
N1 Output	Hard-coded
N2 Output	Only supported if **Output when disabled** is set to “held”
N1 Subsystem	Hard-coded
N1 Switch Case	Hard-coded

Symbol	Note
N1 Switch Case Action Subsystem	Only supported if **States when execution is resumed** is set to “held”.
N2 Switch Case Action Subsystem	Hard-coded
N1 Triggered Subsystem	Does not support **Trigger type** set to “function-call”
N2 Triggered Subsystem	Hard-coded
N1 Width	Hard-coded
N1 Bus Creator	Hard-coded
N1 Bus Selector	Hard-coded
N1 Demux	Hard-coded
N2 Demux	Arity type of output port can not be determined through backward analysis

Symbol	Note
N1 From	Hard-coded
N1 Goto	Hard-coded
N1 Merge	Hard-coded
N2 Merge	Does not support **Allow unequal port widths**
N1 Multiport Switch	Hard-coded
N1 Mux	Hard-coded
N1 Selector	Hard-coded
N1 Constant	Hard-coded

Symbol	Note
N1 Ground	By default translated to a block that output “false”. Integer, real and constant as output is supported to manual selection.
N1 Fcn	Hard-coded

Table 23 – SCADE Drive notes

Appendix VIII – Metrics

STMOB	Code mobility
STCDN	Comment of code ratio
STBME	COCOMO Embedded programmer months
STTDE	COCOMO Embedded total months
STDEV	Estimated development time
STPRT	Estimated porting time
STHAL	Halstead prediction of STTOT
STOPN	Halstead Distinct Operands
STOPT	Halstead Distinct Operators
STECT	Number of External Variables
STFNC	Number of Function Definitions
STSCT	Number of Static Variables
STBMO	COCOMO Organic Programmer Months
STTDO	COCOMO Organic Total Months
STEFF	Program Effort
STVOL	Program Volume
STBUG	Residual Bugs (token-based estimate)
STBMS	COCOMO Semi-detached Programmer Months
STTDS	COCOMO Semi-Detached Total Months
STSHN	Shannon Information Content
STTOT	Total Number of Tokens
STVAR	Number of Identifiers
STTLN	Total Preprocessed Source Lines
STTPP	Total Unpreprocessed Source Lines
STZIP	Zipf Prediction of STTOT
STCYC	Cyclomatic Complexity
STMIF	Maximum nesting of control structures
STPTH	Estimated static path count
STMCC	Myer’s Interval
STSUB	Number of function calls
STFCO	Estimated function coupling
STDIF	Halstead’s program difficulty
STXLN	Number of executable lines
STAKI	Akiyama's Criterion
STKNT	Knot count
STKDN	Knot density
STBAK	Number of backward jumps
STLIN	Number of maintainable code lines
STELF	Number of dangling else-if
STGTO	Number of goto
STLCT	Number of local variables declared
STPDN	Path Density
STPBG	Path-based residual bug estimate
STUNV	Number of unused and unreused variables

A complete explanation of all metrics can be found in the QA C manual[1].

Unscaled

File-based

RTW-EC	STMOB	STCDN	STBME	STTDE	STDEV	STPRT	STFCO	STHAL	STOPN	STOPT	STECT	STFNC	STSCT	STBMO	STTDO	STDIF	STEFF	STVOL	STBUG	STBMS	STTDS	STSHN	STTOT	STVAR	STTLN	STTPP	STZIP
rtwtypes.h	99,67	1,086	0,435	1,916	3,82	0,01	0	993	122	30	0	0	0	0,378	1,727	6,78	22907	3378	1	0,418	1,842	3504	466	77	53	172	851
computersystem.c	99,79	0,704	0,454	1,941	4,81	0,01	0	544	69	26	0	4	0	0,392	1,751	11,12	28863	2595	1	0,434	1,867	1869	395	41	78	178	487
computersystem.h	99,27	1,16	0,214	1,525	0,85	0,01	0	247	33	19	0	0	0	0,203	1,363	5,33	5109	958	0	0,215	1,459	818	168	33	22	95	235
computersystem_private.h	100	1,336	0,071	1,073	0,16	0	0	136	25	7	0	0	0	0,077	0,946	2,82	974	345	0	0,077	1,019	411	69	24	0	38	129
computersystem_types.h	100	2,66	0,041	0,899	0,04	0	0	22	2	7	0	0	0	0,048	0,787	5,15	212	41	0	0,046	0,851	59	13	2	1	24	25
rtlib.h	100	0	0,006	0,492	0,48	0	0	48	7	9	0	0	0	0,009	0,421	10,09	2907	288	0	0,008	0,46	149	72	7	0	5	54
TargetLink																											
tl_types.h	100	7,132	0,031	0,822	0,02	0	0	12	1	5	0	0	0	0,037	0,717	5,44	141	26	0	0,035	0,777	29	10	1	0	19	14
computersystem.c	99,81	3,61	1,074	2,558	10,67	0,02	0	639	79	29	0	2	0	0,833	2,332	18,55	64016	3452	2	0,97	2,474	2223	511	34	142	365	568
computersystem.h	95,31	9,837	0,208	1,513	0,13	0,01	0	67	11	9	7	0	0	0,198	1,352	4,04	786	194	0	0,21	1,447	206	45	11	8	93	71
tl_basetypes.h	100	2,514	0,049	0,954	0,27	0	0	97	11	15	0	0	0	0,056	0,837	5,76	1599	277	0	0,055	0,904	306	59	11	10	28	100
tl_defines_b.h	100	7,454	0,045	0,927	0,05	0	0	13	3	4	0	0	0	0,052	0,813	5,97	285	48	0	0,05	0,878	40	17	2	0	26	18
SCADE																											
scade_types.h	100	11,02	0,051	0,967	0	0	0	5	1	3	0	0	0	0,058	0,849	2,1	21	10	0	0,057	0,917	14	5	1	0	29	8
computersystem.c	89,14	0,145	0,519	2,026	18,32	1,99	0	663	86	24	30	3	0	0,441	1,831	19,99	1E+05	5500	2	0,492	1,95	2280	811	47	167	199	581
computersystem.h	97,01	1,467	0,126	1,287	0,25	0,01	0	139	19	15	0	0	0	0,127	1,142	4,18	1509	361	0	0,131	1,227	452	71	18	15	61	140
definitions.h	100	2,749	0,051	0,967	0,36	0	0	91	13	12	0	0	0	0,058	0,849	6,36	2156	339	0	0,057	0,917	289	73	12	2	29	95

[1] Programming Reseach Quality Assurance C Technical Manual

Table 24 – File-based metrics for unscaled model

Function-based

Real-Time Workshop Embedded Coder		STAKI	STCYC	STMIF	STPTH	STKNT	STKDN	STMCC	STBAK	STLIN	STELF	STXLN	STSUB	STGTO	STLCT	STPDN	STPBG	STUNV
computersystem.c	computersystem_initialize	2	2	1	2	0	0	0	0	14	0	4	0	0	0	0,5	0	0
computersystem.c	computersystem_step	8	8	2	54	0	0	9	0	118	1	41	0	0	9	1,32	2	0
computersystem.c	computersystem_terminate	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
TargetLink																		
computersystem.c	computersystem	19	19	2	34992	0	0	0	0	232	0	77	0	0	14	454,4	5	0
SCADE																		
computersystem.c	computersystem_init	1	1	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0
computersystem.c	computersystem	19	19	1	1E+05	0	0	0	0	161	0	143	0	0	0	1031	5	0

Table 25 – Function-based metrics for unscaled model

Scaled

File-based

RTW-EC	STMOB	STCDN	STBME	STTDE	STDEV	STPRT	STFCO	STHAL	STOPN	STOPT	STECT	STFNC	STSCT	STBMO	STTDO	STDIF	STEFF	STVOL	STBUG	STBMS	STTDS	STSHN	STTOT	STVAR	STTLN	STTPP	STZIP
rtwtypes.h	99,67	1,134	0,423	1,899	3,73	0,01	0	947	118	28	0	0	0	0,369	1,711	6,93	22408	3235	1	0,407	1,825	3324	450	73	49	168	812
computersystem.c	99,75	0,622	0,528	2,038	8,49	0,02	721	87	32	0	5	0	0,448	1,842	13,44	50959	3792	1	0,5	1,962	2530	550	48	93	202	637	
computersystem.h	99,17	1,444	0,241	1,585	0,81	0,01	0	247	33	19	0	0	0	0,225	1,419	5,21	4868	935	0	0,24	1,518	818	164	33	22	105	235
computersystem_private.h	99,84	0,346	0,123	1,279	1,92	0	0	552	77	17	0	0	0	0,125	1,135	5,18	11508	2222	1	0,128	1,219	1843	339	68	1	60	481
computersystem_types.h	100	2,667	0,041	0,899	0,04	0	0	22	2	7	0	0	0	0,048	0,787	5,15	212	41	0	0,046	0,851	59	13	2	1	24	25
rtlib.h	100	0	0,006	0,492	0,48	0	0	48	7	9	0	0	0	0,009	0,421	10,09	2907	288	0	0,008	0,46	149	72	7	0	5	54
TargetLink																											
tl_types.h	100	7,132	0,031	0,822	0,02	0	0	12	1	5	0	0	0	0,037	0,717	5,44	141	26	0	0,035	0,777	29	10	1	0	19	14
computersystem.c	99,73	4,048	1,039	2,531	15,7	0,04	0	728	88	32	0	2	0	0,809	2,307	21,38	94203	4407	2	0,941	2,447	2561	638	37	114	355	644

computersystem.h	95,31	10,858	0,208	1,513	0,13	0,01	0	67	11	9	7	0	0	0,198	1,352	4,04	786	194	0	0,21	1,447	206	45	11	8	93	71
sat2.h	100	0	0,021	0,731	1,04	0	0	62	10	9	0	0	0	0,027	0,635	12,05	6243	518	0	0,025	0,689	192	122	10	0	14	67
tl_basetypes.h	100	2,514	0,049	0,954	0,27	0	0	97	11	15	0	0	0	0,056	0,837	5,76	1599	277	0	0,055	0,904	306	59	11	10	28	100
tl_defines_b.h	100	7,454	0,045	0,927	0,05	0	0	13	3	4	0	0	0	0,052	0,813	5,97	285	48	0	0,05	0,878	40	17	2	0	26	18
SCADE																											
scade_types.h	100	2,865	0,087	1,145	0,07	0	0	36	8	5	2	0	0	0,092	1,012	3,45	396	115	0	0,093	1,089	108	31	8	6	45	41
computersystem.c	64,98	0,473	1,297	2,717	12,19	4,27	48	1040	138	15	56	3	0	0,982	2,483	9,12	73152	8019	2	1,157	2,631	3536	1105	124	301	427	858
computersystem.h	96,92	1,974	0,128	1,295	0,2	0,01	0	134	18	15	0	0	0	0,129	1,15	3,73	1206	323	0	0,133	1,235	430	64	18	15	62	134
computersystem_extern.h	100	1,73	0,076	1,094	0,2	0	0	85	19	3	19	0	0	0,082	0,965	3,63	1213	334	0	0,082	1,04	240	75	19	15	40	81
config_types.h	100	0	0,009	0,56	0,03	0	0	31	9	2	0	0	0	0,013	0,482	2,43	168	69	0	0,012	0,525	83	20	7	0	7	33
definitions.h	100	5,11	0,047	0,941	0,35	0	0	67	9	11	0	0	0	0,054	0,825	7,38	2074	281	0	0,053	0,891	206	65	9	2	27	71
macro_default.h	71,33	1,658	0,672	2,202	8,61	2,47	0	249	39	12	0	0	0	0,553	1,996	15,34	51684	3369	1	0,627	2,123	798	594	39	15	247	230
macro_libimplementation_user.h	100	0	0,018	0,689	0	0	0	0	0	0	0	0	0	0,023	0,597	0	0	0	0	0,021	0,649	0	0	0	0	12	0

Table 26 - File-based metrics for scaled model

Function-based

RTW-EC		STAKI	STCYC	STMIF	STPTH	STKNT	STKDN	STMCC	STBAK	STLIN	STELF	STXLN	STSUB	STGTO	STLCT	STPDN	STPBG	STUNV
computersystem.c	computersystem_initialize	2	2	1	2	0	0	0	0	9	0	3	0	0	0	0,67	0	0
computersystem.c	computersystem_step	9	8	2	54	0	0	9	0	132	1	45	1	0	8	1,2	2	0
computersystem.c	computersystem_terminate	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
computersystem.c	div_s16s32	2	2	1	2	0	0	4	0	12	0	8	0	0	2	0,25	0	0
TargetLink																		
computersystem.c	computersystem	13	13	2	1296	0	0	3	0	218	0	61	0	0	13	21,25	3	0
SCADE																		
computersystem.c	computersystem_init	1	1	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0
computersystem.c	computersystem	69	19	1	262144	0	0	0	0	384	0	247	50	0	28	1061,31	5	0

Table 27 – Function-based metrics for scaled model

Appendix IX – MISRA C compliance

The tables below are copies of the compliance list provided by the software providers.

MISRA C: 1998 RULE Number	Rule Class	Possible Simulink violations	Addressing violations with model style	Comments	Simulink Compliance Possible
1	Required	None		Code is per the standard.	Yes
2	Advisory	None		Other languages are not used.	Yes
3	Advisory	None		Assembly language is not generated.	Yes
4	Advisory	Divide By zero case and math equations resulting in overflow.	Simulink and Stateflow provide diagnostics to detect run-time errors in model. The diagnostics are part of the Configuration Parameters dialogue. When the diagnostics are set to "error", errors are detected and reported (upon model simulation) . It is advisable to simulate the model before code generation to catch run-time errors. Some of the Diagnostics provided are - Data Overflow - Array bounds exceeded - Inf or Nan Block output (detects divide by zero)	Users can explicitly model run-time checks. For e.g. special 'Protected Divide' block can be modelled to check for divide by zero.	Yes
5	Required	None		Nothing out of the ordinary is used.	Yes
6	Required	None			Yes

7	Required	None		Not used.	Yes
8	Required	None		MISRA 2004 has removed this rule.	Yes
9	Required	None			Yes
10	Advisory	None			Yes
11	Required	Signals/parameters/states can violate the rule if labelled with more than 31 character	Use Configuration Set parameters to control identifiers - Set the maximum identifier length to 31. - Use model names less than 17 chars. - Allow Mangling in the Symbol string and avoid model name in the symbol string	User is responsible for checking compiler/linker for 31 character support. RTW does not control the length of user-defined identifiers. Please see Rule 11 assumptions at the bottom of the spreadsheet.	Yes.
12	Advisory	User-specified identifiers are not checked for reuse. For e.g a subsytem function name speciifed by the user can be same as a 'ExportedGlobal' signal name	Do not reuse names in different name spaces. User can track all usage the same way it would be done with manual methods. Scripts can also be written to help.	This requires that name unique control is turned on.	Yes
13	Advisory	None	None	Both RTW and SF redefine the base types using "rtwtypes.h".	Yes

14	Required	Plain char used for storing character values	None	RTW uses plain char for storing character values. MISRA C 2004 states that character strings need not be declared as signed/unsigned char	No
15	Advisory	None			Yes
16	Required	None		No usage of bit representation.	Yes
17	Required	None			Yes
18	Advisory	"L" for long is not supported. Assume applies only to base10 constants.	Do not define data that is "long"	MISRA 2004 has removed this rule.	Yes
19	Required	None			Yes
20	Required	None		RTW uses external function declarations. SF Coder uses function declarations just before the function definition (body). MISRA 2004 has removed this rule.	Yes
21	Required	None		Assume that inner scope/outer scope refer to block statements within functions. It is assumed that this rule does not apply to global variables with the same names as local variables. If assumption is not correct, then rule is violated.	Yes

22	Advisory	None		RTW has "Local Blocks Output" option that provides function definition of objects. In Stateflow, use graphical functions and "Temporary" Data for defining non-persistent chart data	Yes
23	Advisory	None		By default, RTW treats non-function scope vatiables as globals with extern declarations. Use Custom storage classes to specify 'File' scope on signals and parameters.	Yes
24	Required	None			Yes
25	Required	None			Yes
26	Required	None			Yes
27	Advisory	None			Yes
28	Advisory	None		It is not used. MISRA 2004 has removed this rule.	Yes
29	Required	None			Yes
30	Required	None		Initialization takes place either at object definition or via an initialization function. Users can optionally not genrate explicit zero initialization of data	Yes

31	Required	None		Stateflow will initialize each cell individually therefore avoiding any compiler ambiguity. Simulink matrices, 2-D, and n-D tables are implemented as 1-D arrays and are initialized as 1-D arrays.	Yes
32	Required	None		Enumerated lists are currently not supported.	Yes
33	Required	None	User needs to avoid calling functions with coupled side effects.		Yes
34	Required	None			Yes
35	Required	None			Yes
36	Advisory	None			Yes
37	Required	Fixed-point math operations can introduce bit-shifts and bitwise-AND on signed integers. Simulink BITWISE operator block does not allow for signed INTs as inputs.	Do not use BITWISE operators on signed integers in Stateflow. To avoid bit-shifts in fixed-point math operations, uncheck the option " Shift right on signed integer as arithmetic shift" in Configuration Paramaters -> Hardware Implementation . To avoid bit-wise AND (&), round integers to floor instead of zero.	Some fixed-point operations can have bit-wise operation irrespective of the model settings and correct modeling style suggested in the adjacent column.	No
38	Required	Shifts too far are possible based upon the scaling selected by the user.	Break equations into components that do not require large shifts	User needs to practice good fixed point math rules while modeling.	Yes

39	Required	Violation can occur based upon model style.	User needs to not use unary minus operation on signed expressions in modeling.	Violation can occur based upon model style.	Yes
40	Advisory	None		sizeof is not used.	Yes
41	Advisory	None		This rule is not dependent upon the code generator. It is the user responsibility to understand the complier and microprocessor selected.	Yes
42	Required	None			Yes
43	Required	None			Yes
44	Advisory	Math equations involving mixed data types.	Not easily done	RTW will cast expressions whenever appropriate. However, extra casts may appear in some cases. MISRA 2004 has removed this rule.	No
45	Required	Yes			Yes
46	Required	The usage of a single data store write in two different atomic subsystems will cause a problem.	Avoid side effects associated with functions that may not be executed in order expected during modeling. If using global data with reusable functions, set code reuse error diagnostic to 'Error'	Generally not an issue. However, it is possible to model in such a way to violate the intent of the rule.	Yes
47	Advisory	None.	Break Stateflow mixed operator action statements into separate equations using temporary variables.	Stateflow will depend upon C operator precedence with correct code. RTW uses parenthesis when different operators are present.	Yes

48	Advisory	None		RTW and SF do not analyze type of inputs for mixed precision math. Fixed point is ok	Yes
49	Advisory	None			Yes
50	Required	Simulink Blocks which test for equality do not check for floating point variables.	Cast floating point to integer prior to equality testing.	Both RTW and SF will attempt to test for equality on floating point variables based upon user models. For non-finite number support, RTW will generate rtIsInf() and rtIsNaN() functions. These functions compare two floating point variables. To achieve MISRA compliance, un-check "non-finite support" in model's configuration parameters dialogue.	Yes
51	Advisory	Subtracting a lager unsigned constant from a smaller unsigned constant.	Do not model using a difference between two unsigned constants unless the result will be positive.	Stateflow optimization will not permit this to occur. Wrap around will be determined correctly by code generator.	Yes
52	Required	None		Assume that usage of parameters or constants in predicate equations to turn code segments is not included in the rule.	Yes
53	Required	None			Yes
54	Required	None			Yes

55	Advisory	None	Only use structured patterns that are goto free	Labels in SF are due to GOTOs. MISRA 2004 has removed this rule.	Yes
56	Required	None	Only use structured patterns that are goto free.	Unstructured Stateflow diagrams will produce goto.	Yes
57	Required	None			Yes
58	Required	None		MISRA 2004 has removed this rule.	Yes
59	Required	None			Yes
60	Advisory	An else statement is not generated for a IF block when "show else" switch is turned OFF.	In Simulink, set show else switch must be ON in IF block. In Stateflow, user can graphically add in extra else clause.	Code generator optimizes else statements if they are empty or conditionally not executed.	Yes.
61	Required	None		Further review needed	Yes
62	Required	None			Yes.
63	Advisory	None		Simulink switch block reports error if boolean is used for switch input	Yes
64	Required	None			Yes
65	Required	None	Avoid floating point counters in StateFlow loops		Yes
66	Advisory	None			Yes

67	Advisory	None	Do not model the loop index modification in the for loop. Control with style		Yes
68	Required	None			Yes
69	Required	None			Yes
70	Required	None	Use directed events or graphical functions. Do not use broadcast.		Yes
71	Required	None		RTW Prototypes in a header file. SF prototypes are located next to function unless exported. Assumes rms scheduler not included	Yes
72	Required	None			Yes
73	Required	None			Yes
74	Required	None			Yes
75	Required	None		Either void or a specific type is always used.	Yes
76	Required	None			Yes
77	Required	None			Yes
78	Required	None			Yes
79	Required	None		MISRA 2004 has removed this rule.	Yes
80	Required	None		MISRA 2004 has removed this rule.	Yes

81	Advisory	None		Not an issue since pass by reference is not support except for output of subsystems in reusable subsystems.	Yes
82	Advisory	None		All functions will have a primary single exit at the end of the function. Of course, if branches are required that represent termination, then a return will be present at each branch as permitted in MISRA rule 83.	Yes
83	Required	None		All branches have proper returns	Yes
84	Required	None		MISRA 2004 has removed this rule.	Yes
85	Advisory	None			Yes
86	Advisory	None			Yes
87	Required	None			Yes
88	Required	None			Yes
89	Required	None			Yes
90	Required	None			Yes
91	Required	None	None	Usage is generally to limit scope of define statement. There is no attempt to reuse or overwrite defines.	Yes
92	Advisory	None		User is responsible for checking #undef in custom code	No

93	Advisory	Fixed- point Arithmetic involving non-native integer sizes on a target can result in function-like macros. For e.g operations on a 24bit integer in a 16-bit target	Use native integer sizes	RTW-EC uses functions instead of function-like macros for fixed point arithmetic	Yes
94	Required	None			Yes
95	Required	None			Yes
96	Required	None			Yes
97	Advisory	None			Yes
98	Required	None			Yes
99	Required	None			Yes
100	Required	None			
101	Advisory	None			Yes
102	Advisory	None			Yes
103	Advisory	None			Yes
104	Required	None		MISRA 2004 has removed this rule.	Yes
105	Required	None		MISRA 2004 has removed this rule.	Yes
106	Required	None			Yes
107	Required	None		MISRA 2004 has removed this rule.	Yes
108	Required	None			Yes
109	Required	None	Configuration options. Select Local Block Outputs, etc		Yes
110	Required	None			Yes

111	Required	None		Stateflow can use bit for states. They are unsigned int.	Yes
112	Required	None			Yes
113	Required	None		MISRA 2004 has removed this rule.	Yes
114	Required	Possible to use reserved words for signals/parameters and other identifiers	Avoid reserved words to label signals, parameters, states, functions and other identifiers in the model.	Reserved words such as "float" are detected but library names such as "sprintf" can be used as signal names.	Yes
115	Required	Possible to use library names such as "sprintf" for signals/parameters and other identifiers	Avoid library names to label signals, parameters, states, functions and other identifiers in the model.	RTW does not use library names for temporary variables or model functions.	Yes
116	Required	None		User is responsible for checking compiler libraries and other third-party libraries	Yes
117	Required	No	User can model specific checks for library function calls that are "unknown".	Possible to call c libraries without checks.	Yes
118	Required	None		Dynamic allocation is not used.	Yes
119	Required	None			Yes
120	Required	None			Yes
121	Required	None		setlocale not used. MISRA 2004 has removed this rule.	Yes
122	Required	None			Yes
123	Required	None			Yes
124	Required	stdio.h is included if ert_main.c or if MAT-file logging is selected.	Do not select generate main option or .mat file for RTW-EC.	RTW-EC Generate main option will include stdio.h for printf and flush. Also occurs if .mat file is generated.	Yes

125	Required	None			Yes
126	Required	None		Library functions abort, exit, getenv and system not used.	Yes
127	Required	None		time handling not used.	Yes

Table 28 – RTW-EC compliance with the rules of MISRA C

Rules	SCADE Compliance
Rule1.1[required]	Derivable compliance
Rule1.2[required]	Full compliance
Rule1.3[required]	Not Applicable
Rule1.4[required]	Not Applicable
Rule1.5[advisory]	Not Applicable
Rule2.1[required]	Full compliance
Rule2.2[required]	Full compliance
Rule2.3[required]	Configurable compliance
Rule2.4[advisory]	Full compliance
Rule3.1[required]	Full compliance
Rule3.2[required]	Full compliance
Rule3.3[advisory]	Not Applicable
Rule3.4[required]	Full compliance
Rule3.5[required]	Full compliance
Rule3.6[required]	Not Applicable
Rule4.1[required]	Full compliance
Rule4.2[required]	Full compliance
Rule5.1[required]	Configurable compliance
Rule5.2[required]	Full compliance
Rule5.3[required]	Derivable compliance
Rule5.4[required]	Full compliance
Rule5.5[advisory]	Derivable compliance
Rule5.6[advisory]	Derivable compliance
Rule5.7[advisory]	Full compliance
Rule6.1[required]	Derivable compliance
Rule6.2[required]	Full compliance
Rule6.3[advisory]	Full compliance
Rule6.4[required]	Full compliance
Rule6.5[required]	Full compliance
Rule7.1[required]	Full compliance

Rule8.1[required]	Full compliance
Rule8.2[required]	Full compliance
Rule8.3[required]	Full compliance
Rule8.4[required]	Full compliance
Rule8.5[required]	Full compliance
Rule8.6[required]	Full compliance
Rule8.7[required]	Full compliance
Rule8.8[required]	Derivable compliance
Rule8.9[required]	Full compliance
Rule8.10[required]	Configurable compliance
Rule8.11[required]	Configurable compliance
Rule8.12[required]	Full compliance
Rule9.1[required]	Derivable compliance
Rule9.2[required]	Full compliance
Rule9.3[required]	Full compliance
Rule10.1[required]	Configurable compliance
Rule10.2[required]	Configurable compliance
Rule10.3[required]	Full compliance
Rule10.4[required]	Full compliance
Rule10.5[required]	Full compliance
Rule10.6[required]	Configurable compliance
Rule11.1[required]	Full compliance
Rule11.2[required]	Full compliance
Rule11.3[advisory]	Full compliance
Rule11.4[advisory]	Derivable compliance
Rule11.5[required]	Full compliance
Rule12.1[advisory]	Derivable compliance
Rule12.2[required]	Full compliance
Rule12.3[required]	Full compliance
Rule12.4[required]	Derivable compliance
Rule12.5[required]	Derivable compliance

Rule12.6[advisory]	Configurable compliance
Rule12.7[required]	Derivable compliance
Rule12.8[required]	Full compliance
Rule12.9[required]	Derivable compliance
Rule12.10[required]	Full compliance
Rule12.11[advisory]	Full compliance
Rule12.12[required]	Full compliance
Rule12.13[advisory]	Full compliance
Rule13.1[required]	Full compliance
Rule13.2[advisory]	Full compliance
Rule13.3[required]	Derivable compliance
Rule13.4[required]	Full compliance
Rule13.5[required]	Full compliance
Rule13.6[required]	Full compliance
Rule13.7[required]	Derivable compliance
Rule14.1[required]	No compliance
Rule14.2[required]	No compliance
Rule14.3[required]	Full compliance
Rule14.4[required]	Full compliance
Rule14.5[required]	Full compliance
Rule14.6[required]	Full compliance
Rule14.7[required]	Configurable compliance
Rule14.8[required]	Derivable compliance
Rule14.9[required]	Derivable compliance
Rule14.10[required]	Full compliance
Rule15.1[required]	Full compliance
Rule15.2[required]	Full compliance
Rule15.3[required]	Full compliance
Rule15.4[required]	Derivable compliance
Rule15.5[required]	Full compliance
Rule16.1[required]	Full compliance

Rule16.2[required]	Full compliance
Rule16.3[required]	Full compliance
Rule16.4[required]	Full compliance
Rule16.5[required]	Full compliance
Rule16.6[required]	Full compliance
Rule16.7[advisory]	Derivable compliance
Rule16.8[required]	Full compliance
Rule16.9[required]	Full compliance
Rule16.10[required]	Full compliance
Rule17.1[required]	Full compliance
Rule17.2[required]	Full compliance
Rule17.3[required]	Full compliance
Rule17.4[required]	Full compliance
Rule17.5[advisory]	Full compliance
Rule17.6[required]	Full compliance
Rule18.1[required]	Full compliance
Rule18.2[required]	Full compliance
Rule18.3[required]	Full compliance
Rule18.4[required]	Full compliance
Rule19.1[advisory]	Full compliance
Rule19.2[advisory]	Full compliance
Rule19.3[required]	Full compliance
Rule19.4[required]	Full compliance
Rule19.5[required]	Full compliance
Rule19.6[required]	Full compliance
Rule19.7[advisory]	Derivable compliance
Rule19.8[required]	Full compliance
Rule19.9[required]	Full compliance
Rule19.10[required]	Derivable compliance
Rule19.11[required]	Full compliance
Rule19.12[required]	Full compliance

Rule19.13[advisory]	Full compliance
Rule19.14[required]	Full compliance
Rule19.15[required]	Configurable compliance
Rule19.16[required]	Full compliance
Rule19.17[required]	Full compliance
Rule20.1[required]	Derivable compliance
Rule20.2[required]	Derivable compliance
Rule20.3[required]	Derivable compliance
Rule20.4[required]	Full compliance
Rule20.5[required]	Full compliance
Rule20.6[required]	Full compliance
Rule20.7[required]	Full compliance
Rule20.8[required]	Full compliance
Rule20.9[required]	Full compliance
Rule20.10[required]	Full compliance
Rule20.11[required]	Full compliance
Rule20.12[required]	Full compliance
Rule21.1[required]	Derivable compliance

Table 29 - SCADE compliance with the rules of MISRA C

MISRA Rule	Classification		Compliance				
	required	advisory	Full	Configurable	Partial	Derivable	None
1.1	X				X		
1.2	X					X	
1.3	X		X				
1.4	X		n.a.				
1.5		X	n.a.				
2.1	X			X			
2.2	X		X				
2.3	X		X				
2.4		X	X				
3.1	X						X
3.2	X						X
3.3		X			X		
3.4	X						X
3.5	X		X				
3.6	X						X
4.1	X		X				
4.2	X		X				
5.1	X			X			
5.2	X		X				
5.3	X					X	
5.4	X					X	
5.5		X	X				
5.6		X				X	
5.7		X				X	
6.1	X		X				
6.2	X		X				
6.3		X			X		
6.4	X			X			
6.5	X			X			
7.1	X		X				
8.1	X		X				
8.2	X		X				
8.3	X		X				
8.4	X		X				
8.5	X		X				
8.6	X		X				
8.7	X				X		
8.8	X			X			
8.9	X			X			
8.10	X			X			
8.11	X			X			
8.12	X		X				
9.1	X					X	
9.2	X		X				
9.3	X		n.a.				
10.1	X						X
10.2	X					X	

MISRA Rule	Classification		Compliance				
	required	advisory	Full	Configurable	Partial	Derivable	None
10.3	X						X
10.4	X		X				
10.5	X		X				
10.6	X						X
11.1	X		X				
11.2	X		X				
11.3		X	X				
11.4		X					X
11.5	X						X
12.1		X					X
12.2	X		X				
12.3	X		X				
12.4	X					X	
12.5	X		X				
12.6		X					X
12.7	X						X
12.8	X						X
12.9	X						X
12.10	X		X				
12.11		X	X				
12.12	X		X				
12.13		X					X
13.1	X		X				
13.2		X			X		
13.3	X						X
13.4	X		X				
13.5	X				X		
13.6	X				X		
13.7	X				X		
14.1	X				X		
14.2	X		X				
14.3	X						X
14.4	X		X				
14.5	X		X				
14.6	X		X				
14.7	X					X	
14.8	X				X		
14.9	X				X		
14.10	X						X
15.1	X		X				
15.2	X		X				
15.3	X						X
15.4	X		X				
15.5	X		X				
16.1	X		X				
16.2	X					X	
16.3	X		X				
16.4	X					X	
16.5	X		X				
16.6	X		X				
16.7		X			X		
16.8	X		X				
16.9	X		X				
16.10	X		X				
17.1	X		X				
17.2	X		X				
17.3	X		X				
17.4	X						X
17.5		X	X				
17.6	X		X				

MISRA Rule	Classification		Compliance				
	required	advisory	Full	Configurable	Partial	Derivable	None
18.1	X		X				
18.2	X		X				
18.3	X						X
18.4	X						X
19.1		X	X				
19.2		X			X		
19.3	X		X				
19.4	X				X		
19.5	X			X			
19.6	X			X			
19.7		X					X
19.8	X		X				
19.9	X		X				
19.10	X						X
19.11	X		X				
19.12	X		X				
19.13		X		X			
19.14	X		X				
19.15	X		X				
19.16	X		X				
19.17	X		X				
20.1	X		X				
20.2	X			X			
20.3	X		X				
20.4	X		X				
20.5	X		X				
20.6	X		X				
20.7	X		X				
20.8	X		X				
20.9	X		X				
20.10	X		X				
20.11	X		X				
20.12	X		X				
21.1	X			X			
# 141	121	20	79	13	14	11	24

Table 30 – TargetLink compliance with the rules of MISRA C

Full Compliance	The rule is always met.
Derivable Compliance	In order to fully comply with the rule, certain restrictions at the modellevel have to be observed.
Configurable Compliance	Using some code generator options can configure the generated code. Selecting or cleaning a certain option makes the generated code compliant with the rule.
Partial Compliance	The rule states several requirements, some of which are fulfilled and some of which are not.
No Compliance	The generated code may deviate from the rule.
Not Applicable	Does not depend on the SCADE code generator. This is a compiler issue for instance.

Table 31 – Explanation of the different levels of compliance.