

JUST SIMPLIFY: Clone Detection for Simulink Controller Models

Elke Salecker and Ingo Stuermer

Model Engineering Solutions

ABSTRACT

Huge Simulink controller models often consists of (almost) identical subsystems, very often resulting from copy-and-paste operations and only slight adaptation of the subsystems by the model engineer. Although this “copy-and-paste” approach might help to achieve initial results very fast, in the long-run such subsystem clones can create considerable problems. Like code clones, model clones increase the effort for testing and maintenance. Model clones also tend to influence the code efficiency and code quality in a negative way in case the Simulink model is used as a basis for code generation. JUST SIMPLIFY is an approach for detecting model clones in a Simulink model automatically based on model metrics calculations. This approach has been implemented in our model metrics and complexity measurement tool M-XRAY. JUST SIMPLIFY allows reducing the effort for model refactoring by avoiding time consuming manual search for model clones. As a result, the effort for testing and for maintaining models can be reduced and the code quality can be improved significantly.

CITATION: Salecker, E. and Stuermer, I., "Just Simplify: Clone Detection for Simulink Controller Models," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 9(1):2016, doi:10.4271/2016-01-0026.

1. INTRODUCTION

Model-based development of embedded software with Simulink [1] has become common practice for the development of complex software-based controller functions in the car. This approach allows specifying the functionality of the control algorithm graphically, simulating the model to validate the functionality and finally generating the production code automatically from the model by using a code generator like Embedded Coder [1] from MathWorks or TargetLink from dSPACE [2]. The safety standard ISO 26262 [3] considers this approach as applicable for the development of safety-critical software and states in part 6

The seamless utilization of models facilitates highly consistent and efficient development.

As model-based development has become common practice for the development of safety-critical software all the requirements of ISO 26262 must be implemented for model-based software as well. These requirements include addressing testability and maintainability of the software.

In recent years, the importance of quality of models in general and testability and maintainability of models in particular has increased considerably. One reason is that ISO 26262 requires addressing these aspects. Another reason is the fact that models just as source code must be maintained over a long period. In practice, the maintainability of models is often impaired due to model clones. Model clones are subsystems that are copies of each other. Instead of creating a reusable library, the developer copies a subsystem in order

to implement equal or similar functionality twice or more times. The effort for implementing functional changes and testing is compared to model libraries increased because all copies must be identified and updated or tested.

This problem is not inherent to model-based development and has been known for programming-language based software development for many years. Although for programming languages broad support for refactoring of code clones is available, the situation is quite different for model-based development. Refactoring of models with regard to model clones is currently a manual task for Simulink controller models used for series production.

In this paper, we present our approach JUST SIMPLIFY that is implemented in our model metrics and complexity measurement tool M-XRAY. Our approach detects duplicated subsystems based on model metrics calculations. This eases model refactoring considerably because the time consuming manual search for model clones is no longer necessary. The tool analyses the model and identifies copied subsystems. These subsystems are reported to the developer who uses the analysis results for refactoring the architecture of the model with regard to maintainability and testability. The automatic search for model clones decreases the effort for model refactoring considerably because the number of subsystems in Simulink controller models used for series production can amount to several hundreds.

The remainder of the paper is organized as follows. In we introduce the foundations required to understand our approach. We explain our approach and its implementation in our model metrics and complexity measurement tool M-XRAY in Section 2. In Section 3 we

present evaluation results for applying our clone detection algorithm on Simulink controller models used for series production. In [Section 5](#) we summarize related work and conclude our paper in [Section 6](#).

2. FOUNDATIONS

In this section, we introduce first model metrics because our approach for detecting model clones is based on model metrics. We explain basic metrics and introduce the more precise metrics *local* and *global complexity*. Moreover, we introduce a commonly used classification of model clones which allows for distinguishing between different types of model clones and is used for assessing the capabilities of clone detection algorithms.

2.1. Overview on Model Metrics

Model metrics correspond to code metrics for source code and allow for assessing and comparing models with regard to specific properties such as size or complexity. A model metric analyses a model with regard to the modeling elements it is built of and calculates a numerical value based on this analysis. To assess the model, the actual result value of a metric is compared with a reference value that has been derived from experience. Model metrics are also used to monitor development of a model over time by calculating and comparing model metric values for the different development stages of the model.

Commonly known model metrics are *number of blocks* or *depth of model hierarchy* which basically count model elements. The former counts the number of blocks and block types, the latter the number of levels in the hierarchy of subsystems. These model metrics provide a rough estimation of the size of the model. A more precise metric is the *model volume*. For many metrics, there are two versions: a local and a global metric. The local metric considers a subsystem only, the global metric considers additionally also all the subsystems that are part of it. The metric *local number of blocks* counts only the blocks in a subsystem, the metric *global number of blocks* sums up the blocks in the subsystem and additionally all the blocks of the subsystems that are contained in it.

To illustrate the difference between the basic metrics and the more precise metrics with regard to their results we use the Simulink subsystems shown in [Figure 1](#) and [Figure 2](#). Both subsystems are part of the model *fuelsys* which is part of the Simulink demo library [1].

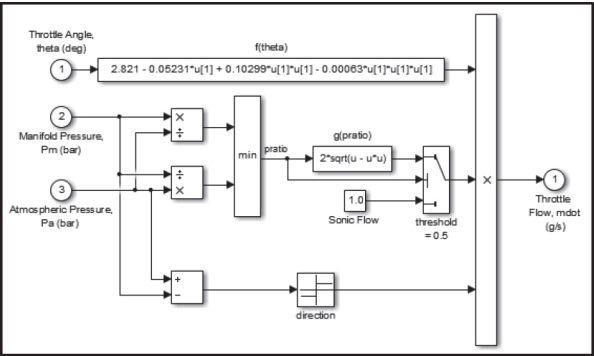


Figure 1. Subsystem *Throttle* that is part of the Simulink demo model *fuelsys*

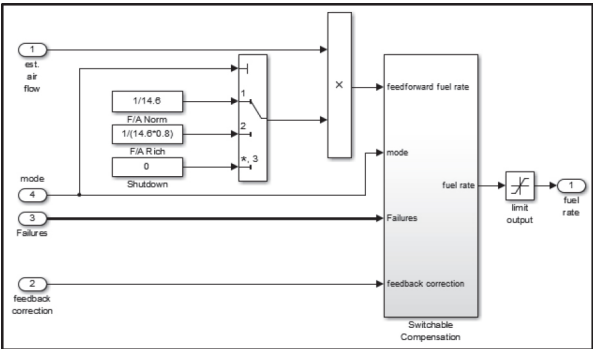


Figure 2. Subsystem *Fuel Calculation* that is part of the Simulink demo model *fuelsys*

[Table 1](#) shows the values for the metrics *number of blocks*, *local* and *global complexity* for the two subsystems. The value for the *number of blocks* metric is almost identical for both subsystems but the value for the *local complexity* differs by a factor of almost two. Comparing both subsystems, we observe that understanding the subsystem *Fuel Calculation* ([Figure 2](#)) is easier compared with understanding the subsystem *Throttle* ([Figure 1](#)). In the subsystem *Fuel Calculation* there are fewer block types used. Moreover, the linkage between the blocks is simpler in this subsystem. For example, there is only one branching signal that is used as input for two blocks (input signal *mode*). In the other subsystem, first, there are more branching signals and second, these branching signals are used as input for several blocks (e.g. input signals *Manifold Pressure*, *Atmospheric Pressure*).

Table 1. Metric values for the subsystems shown in [Figure 1](#) and [Figure 2](#)

Subsystem	Number of blocks	Local complexity	Global complexity
Throttle	14	151	151
Fuel Calculation	12	75	295

The local complexity metrics takes into account the aspects illustrated in our example. Relevant factors influencing the complexity of a Simulink model are the number and the type of blocks used in the model. Another important factor is the linkage between the used blocks. This linkage represents the data-flow within the model. The complexity induced by the data-flow in a Simulink model requires considering, first, the number of signal lines and second the usage of these signals. In our example, we have seen that signals can be used as inputs for several blocks. Moreover, a model can contain feedback loops. Within a feedback loop, signals are stored in order to be used as input in the following execution cycle. Model volume calculation covers all these factors. For this reason, the metric provides more precise results for assessing and comparing models.

2.2. Model Volume Calculation

The *local complexity* metric is an adaptation of the *Halstead volume* metric for source code. This metric counts for a program in a programming language:

- N1: the total number of operators,
- N2: the total number of operands,
- N3: the number of distinct operators, and
- N4: the number of distinct operands.

The metric value is calculated as follows:

$$\text{Halstead Volume} = (N1 + N2) * \log_2(N3 + N4)$$

The adaptation for model volume calculation is based on an obvious mapping of first, source code operators onto block types and second, source code operands onto signals. This leads to the following definitions:

- N1: the weighted number of blocks,
- N2: the weighted number of block inputs,
- N3: the weighted number of block types, and
- N4: the weighted number of block outputs.

Unlike the Halstead metric for source code, our adaptation calculates the values of N1 and N3 as a weighted sum. This calculation is based on the definition of weights for the different blocks. The weight of a block depends on its functionality. For example, the weight of the 'Sum' block is 0.8; the weight of the 'Switch' block is 1.6. The 'Switch' block is classified as more complex than the 'Sum' block. Signals are also treated differently compared to the original Halstead metric. For each block its inputs and outputs are counted followed by a log₂-weighting. This adaptation is based on the observation that understanding the data dependencies is much easier in a model as in a source code program. The model volume is calculated with the formula:

$$\text{Model volume} = (N1 + N2) * \log_2(N3 + N4)$$

In a similar way as the metric *number of blocks* introduced in the beginning of this section, the first factor $(N1 + N2)$ in the formula for model volume calculation counts the number of modelling elements. Because we use block weights for calculating N1 our result is more precise with regard to measuring complexity. The second factor $\log_2(N3 + N4)$, reflects that the complexity of source code and models does not only depend on its 'sheer size', but on its 'diversity' and the 'linkage'. If a subsystem in our model uses only two different arithmetic blocks for a cascading calculation it is obviously less complex than a subsystem using many different blocks connected by many branching signals. The model volume calculation considers 'size', 'diversity' and 'linkage' which allows to provide precise results for complexity assessment.

We illustrate the model volume calculation with the example subsystem show in Figure 3.

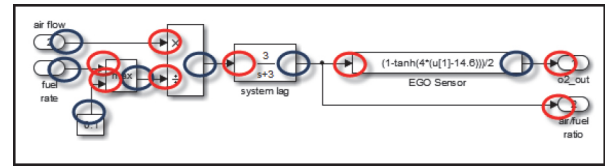


Figure 3. Subsystem *Mixing & Combustion* that is part of the Simulink demo model *fuelsys*

The subsystem *Mixing & Combustion* consists of nine blocks with seven different block types. The values for N1 and N3 are calculated as weighted sums to differentiate for block types. The blue circles indicate the number of different signals (N4). The red circles indicate the use of the signals (N2). For the calculation of N2 and N4 the log₂ weighting is used. The model volume for the example is

$$59 = (9 + 7.13) * \log_2(6.8 + 6.8).$$

The *global complexity* metric is based on the *local complexity* metric. The *local complexity* metric measures the content of a subsystem only, without looking at the functionality encapsulated in subsystems deeper in the model hierarchy. For assessing the subsystem itself together with the functionality implemented by the subsystems deeper in the hierarchy tree, the *global complexity* metric is used. The value of this metric is the sum of the local complexity value of the subsystem and the global complexity values of all the subsystems that it contains.

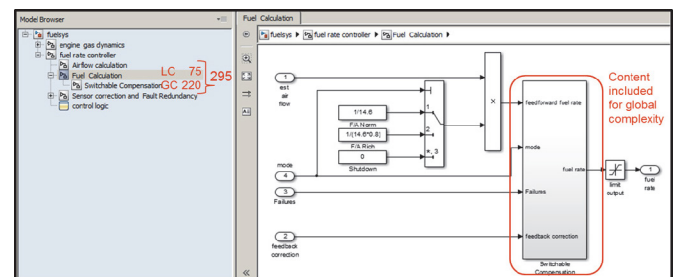


Figure 4. Calculation of the global complexity value for the subsystem *Fuel calculation*

We illustrate the global complexity calculation for the subsystem *Fuel Calculation* (See Figure 2) in Figure 4. The calculation of the local complexity does not take into account the functionality implemented by the subsystem *Switchable Compensation*, i.e., it handles the subsystem block in the same way as the other blocks. In contrast to that, the global complexity value calculation adds the global complexity value 220 of the subsystem *Switchable Compensation* and the local complexity value of 75 of the subsystem itself. For the subsystem *Throttle* shown in Figure 1, the value of the global and local complexity are the same (see Table 1) because this subsystem does not contain any subsystem.

Further details on the calculation of the global and local complexity metrics can be found in [4].