# Modular and Optimized C++ Code-Generator for the Component and Connector Modeling Language MontiCAR

Sascha Schneiders[a], Michael von Wenckstern[a], Evgeny Kusmenko[a], and Bernhard Rumpe[a]

a Chair of Software Engineering, RWTH Aachen, Germany (This paper will be submitted to a journal later on, this is the reson for this specific template layout and the additional authors)

**Abstract** In cyber-physical systems, physical and software components are deeply intertwined and mostly interact with their environment; examples in the automotive industry are: engine control, trajectory planning, steering control and lane correction. Large German automotive manufactors develop CPSs using C&C models, which will be later translated to C/C++ code to deploy it on embedded chips. Testing the C&C models on the hardware specific chips is very expensive and thus is only done at the end of the developing process; in early stages the C&C models are tested inside Simulator frameworks on the developer's computer to give feedback immediately. In test-driven modeling it is very important that modelers can test their changes in a very fast way without waiting several minutes for the modified C&C model to compile, and also the execution of the C&C tests should be done as fast as possible. Therefore, we present in this paper an approach on how to reduce the generation, compilation and execution time of C/C++ code that is created from C&C models.

**Keywords** code generation, model-driven software engineering

The Art, Science, and Engineering of Programming

**Perspective** The Engineering of Programming

**Area of Submission** Code Generation, Model-driven programming

## 1 INTRODUCTION

Challenges in developing Cyber-Physical Systems (CPSs)[16] originate from continuous interactions with their environment. These real-world interactions are subject to physical laws and executed by utilizing sensors and actuators.
It is therefore crucial to test these complex systems. However, performing real-world tests is often very cost and time intensive due to the needed hardware infrastructures. Moreover, a malfunctioning during tests might cause dangerous situations. Therefore, as a first step, tests should be executed in simulation frameworks to validate the behavior of CPSs.
To gain the most benefits out of test-driven development, which means ensuring highly validated software components in CPSs, it is necessary that developers can execute their module tests to validate their current changes at any time and receive immediate feedback whether their components are still working as expected or not. In general, CPSs are modern systems which require a continuous interaction with their surroundings and are usually used to perform safety critical operations such as emergency braking when detecting pedestrians or controlling a chemical reactor. In both cases, the CPS must be operating in real-time and correctly, as acting too late, or performing an incorrect action can result in hazardous consequences. To avoid this, a CPS must be developed with care. To increase robustness, security and early detection of potential errors without testing, programming languages which allow checking a lot of logical errors, like matrix dimension misfits, at compile-time, are required. Simulink[18] which is one of the de facto standards of software used to develop CPSs in the industry, is a splendid example of how the development process of CPSs can be supported but it also shows that there are still improvements to be made. These improvements include the minimization of possible runtime errors by increasing compile-time checks.
As different devices must react to a lot of information, which includes camera sensors, several performance heavy tasks need to be performed, in this case image recognition. As this involves a lot of matrix addition and multiplication the need for optimizing all tasks that utilize many mathematical operations arises. At first we will present two different examples which will be used to present different cases where the automatic optimizations of our generator are applicable. One example is spectral clustering for image recognition, and the other example presents an abstract C&C model which performs some common matrix operations. The MontiCAR[15] language family includes the MontiMath language, which is used to describe the behaviour of a component in a C&C model. An introduction to the most important preliminaries when dealing with MontiCAR, Octave, Armadillo and optimizations the generator performs in general is given in section 3. As MontiMath contains complex mathematical expressions like matrix addition, multiplication and inversion, the ability to utilize an already existent math library in the generated C/C++ code is a huge advantage, as no new C++ math library has to be developed. For C++ there exist different mathematical frameworks like Octave[5] and Armadillo[23]. The modularity of the code generator is increased by separating the handling of mathematical commands from the rest of the generator implementation. When the handling of mathematical

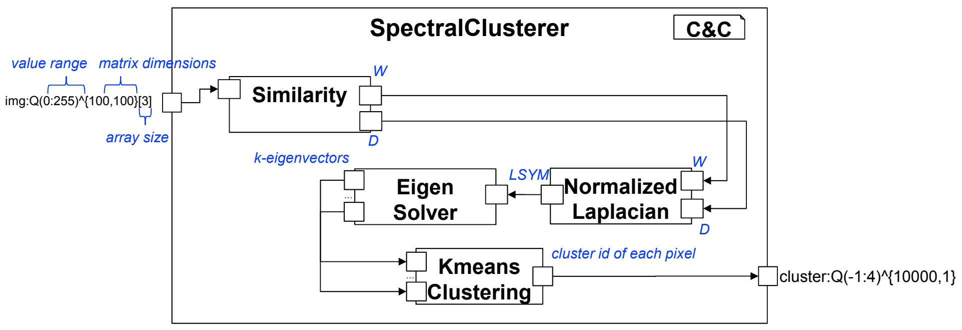

**Figure 1** C&C architecture of the SpectralClusterer.

commands can be easily changed, supporting different additional backends takes less time.
MontiCAR is the language family we use as the source language for our C/C++ code generator. Octave and Armadillo are math libraries which can be linked to different backends which handle the BLAS operations. Available BLAS backends include OpenBLAS[29] and Intel MKL[10]. Not only an efficient usage of Octave or Armadillo, but also the separation of independent parts of software into different threads, when appropriate threading capabilities are available, is important. The used algebraic and threading optimizations are explained in section 4. An explanation of an incremental generation process will be given in section 5. The actual measurable performance gains are evaluated in a case study in section 6.

## 2 RUNNING EXAMPLES

Two running examples are used to examine our approach of a modular and optimized code generator. The first one is an ObjectDetector which uses spectral clustering for image recognition and the second one is a C&C model that executes different matrix calculations which we call MatrixModifier. In praxis map interpolation and transformations which use large matrices are a common case where these kind of operations might occur.
Unsupervised learning has proven to be an important set of tools for automated data understanding and pre-processing. One prominent application is image segmentation, e.g., for the camera input of a self-driving car to separate objects in the scene. One of the most successful approaches are the so called spectral clustering algorithms [21][26]. Let $x_{ij} \in [0,255]^3$ be the 3-dimensional value of a pixel of an image at position $(i,j)$ (e.g. HSV). First, a symmetric similarity matrix $W \in \mathbb{R}^{N\times M}$ where $N$ and $M$ are the height and the width of the image, respectively, has to be computed. Consequently, the entry of $W$ at position $(i,j)$ provides information about how similar two pixels are. Similarity might be computed in terms of distance, color, gradients etc. Second, the so called graph Laplacian is computed as $L = D - W$ and $D$ is the so called degree matrix defined as $D = \operatorname{diag}(W \mathbb{1}_{N\times M})$ where $\mathbb{1}_{NM}$ is an $N \times M$ dimensional column vector full of ones. Often it is advantageous to use the symmetric Laplacian

$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ as outlined in [26]. Note that this step requires a matrix inversion on the diagonal matrix $D$ and two matrix multiplications. Now the first $k$ eigenvectors of $L_{sym}$ have to be computed where $k$ is the number of clusters we want to detect. If this number is unknown an index can be used to estimate it [4]. Furthermore, let $U$ be a $NM \times k$ matrix with the $k$ eigenvectors as its columns. Each row of this matrix represents one pixel in a feature space which should be easier to cluster by the standard k-means algorithm. Finally, the ObjectDetector can separate the objects shown in figure 2 as depicted in figure 3. It does so by using a SpectralClusterer which is depicted in figure 1. The SpectralClusterer is based on the Matlab[19] implementation of the spectral clustering algorithm from Ng, Jordan and Weiss as presented in [26], which was created by Asad Ali and can be downloaded at https://goo.gl/JvUwWA. This implementation was only slightly modified to support loading an image and using it as input data and was then ported to MontiCAR.

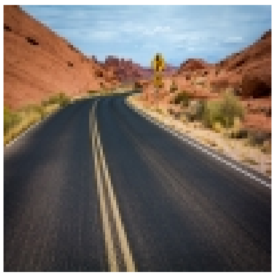

**Figure 2** Original image before using spectral clustering, modified and taken from http://finda.photo/.

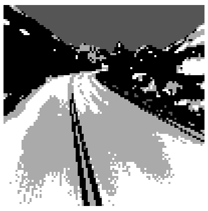

**Figure 3** Resulting image after spectral clustering. It can be seen that the street is clearly separated from its surrounding by using 4 clusters.

The second example depicted in figure 9 shows a component which does several common matrix modification operations. To better outline the performance benefits which are gained by using a smart code generator, this example is chosen to be rather abstract but quite computation intense when the code is generated naively. Similar models are usually utilized inside of applications used in computation intense areas where a lot of information is stored and modified in matrices in different components of the model. Further detail will be provided in the respective sections.
The interface of the MatrixModifier component is defined by its input and output ports. On the left side of the image are the input ports of the model which are used inside of the model in different operations like addition and multiplication. On the right side is the output port which represents the resulting matrix that is calculated by the model. Since it is a conceptual example, no specific input values are examined, as

the amount of operations required is the used metric which determines the speed of an algorithm and in our case also the speed of the generated code in general. The amount of operations for a matrix addition $A + B$ for two matrices A and B with dimensions $n \times m$ is estimated as $2nm$, for matrix multiplication $A \cdot B$ for two matrices A with dimensions $n \times m$, and B with dimensions $m \times p$ the amount of operations is estimated as $2nmp$ [28]. The input ports mat1, mat2, mat3 and mat4 all have a matrix with certain dimensions as type, except the factor input port, which has a scalar of a value, between 0 and 1(included), as type. The full source of both examples can be found at https://goo.gl/gvaa79.

## 3 PRELIMINARIES

### 3.1 C&C Modeling and MontiCAR

When doing model-driven software development, the focus is put onto functionality. This can result in poor performance when no proper code generation tool exists that generates efficient code out of the created models. MontiCAR is a textual C&C based modeling language family that can be used to describe simple and complex models. The basic features which are most relevant for the generator optimization are explained in this section which is basically the EmbeddedMontiArcMath language(see [15] for a detailed description of the language). It consists of components which have input and output ports that are connected by using connectors.

Note that a component may also have subcomponents which receive input from the main component and the output of the subcomponent can be used to affect the output of the top component. This basic C&C concept is enriched by an integrated math language. As MontiCar is strongly typed, errors like wrong matrix dimensions are caught at compile-time, in contrast to Simulink where this is a run-time exception. A matrix can be typed diagonal which allows performance optimizations of the generated code. This is optional but important type information like the matrix dimension has to always be specified. Figure 6 shows how the ObjectDetector example is defined in MontiCar. The textual model starts with an optional package declaration where the package of the component has to be specified, to allow the same component names in different packages. Then import statements can be used to import components from other packages, or whole packages similar to how it works in the Java [1] programming language. The import section is then followed by the keyword *component* and the name of the component. Optional configuration parameters can be provided inside brackets, separated by a commata. Inside curly brackets the body of the component is described. This includes but is not limited to ports, connectors, subcomponents and the *implementation Math* section. As the generator directly operates on the created symboltable after the symbol creation step in the compiler, the important symbols are explained in the following. In general the most relevant symbols for the optimization process of the generator are component, port, connector and expression symbols. A component symbol has a reference to all its subcomponents, ports, connectors, and expressions that are used in the *implementation Math* section. The port symbol contains information like the name and type of the port. A connector stores the name of both ports that are connected. The expression symbols contain information related to the type of expression which could be an assignment and its content, which could be the name of the variable that is assigned a certain value. It is important to note that inside of the *implementation Math* section, each variable has to be initialized, otherwise its values are considered to be not defined/used. This is essential when looking at matrix operations. If an expression $Q\hat{}\{n,m\}\ mat;$ is meant to be a matrix filled with zeros it has to be written as $Q\hat{}\{n,m\}\ mat = zeros(n,m);$. In figure 7 the *implementation Math* section of the `KMeansClustering` component is enriched with comments to explain each line.

### 3.2 Octave and Armadillo

GNU Octave[5] itself is a high-level language and framework which supports efficient and effective methods to perform numerical computations. It also has a C++ interface, which allows using the functionality of Octave in C++ applications.

Armadillo[23] is an open source C++ linear algebra library and can also be used as a math backend in our generator.

The output code of the current generator version uses Octave version 4.2.1 or Armadillo version 8.200.2.
Octave itself relies on a back-end for executing Basic Linear Algebra Subprograms(BLAS). BLAS itself is a specification of low-level routines which perform basic numerical operations, like vector addition, vector multiplication or matrix multiplication. There are several different BLAS back-ends available for usage in Octave. This includes but is not limited to Intel® Math Kernel Library[27] and OpenBLAS[29]. OpenBLAS is used as the BLAS implementation for Octave in the evaluation in section 6. Armadillo also uses BLAS instructions. Therefore, it can also be linked to different BLAS backends and benefit from these optimized implementations. In section 6 the optimizations gains when utilizing Armadillo or Octave are evaluated. This includes the following optimizations:

- Threading different operations which are parallelizable.
- Rewriting terms that include numerical operations like matrix addition and matrix multiplication to increase performance.
- Using different mathematical operations which yield the same result. An example would be the computation of the inverse of a diagonal matrix.

## 4 OPTIMIZATIONS

In this section we present two important optimizations for the to be generated code of our generator, algebraic and threading optimizations. The optimizations of algebraic operations are more important the more mathematical operations are used in the to be generated code based on an input model, while threading optimizations are increasingly important the more cores a CPU has available for computation.

### 4.1 Algebraic Optimizations

The algebraic optimizations that are applicable by the generator which are related to the running examples will be explained in the following. Figure 8 shows the NormalizedLaplacian component which is an inner component of the SpectralClusterer component. It contains the equation 1 which includes the inversion of the degree matrix.

$$\mathrm{nLaplacian} = \mathrm{degree}^{-0.5} \cdot \mathrm{W} \cdot \mathrm{degree}^{-0.5} \tag{1}$$

The degree matrix is derived from the Similarity component, as it is the output port of that component.

This matrix is a diagonal matrix which means that the inverse of this matrix is calculated by replacing every element $e$ on the diagonal by $1/e$.

When examining the SpectralClusterer component in Figure 1, the following properties can be observed by the generator:

- The NormalizedLaplacian component which is an inner component of the Spectral-Clusterer and can be seen in figure 8 uses $\mathrm{degree}^{-0.5}$ twice, therefore it can be calculated only once and the result reused.
- The inversed matrix is a diagonal matrix.

These properties can then be used to generate more efficient code.
Now the MatrixModifier example is examined closer. We implement different optimization when utilizing Octave or Armadillo. They include but are not limited to the following elementary matrix properties [8] with matrices $\mathrm{A}, \mathrm{B}, \mathrm{C}$, vector $\mathrm{b}$ and scalar $\lambda$ appropriately chosen so that the depicted operations are well defined:

$$\mathrm{AC} + \mathrm{BC} = (\mathrm{A} + \mathrm{B})\mathrm{C} \tag{2}$$

$$\mathrm{CA} + \mathrm{CB} = \mathrm{C}(\mathrm{A} + \mathrm{B}) \tag{3}$$

$$\mathrm{A}(\mathrm{BC}) = (\mathrm{AB})\mathrm{C} \tag{4}$$

$$\mathrm{A}(\mathrm{B} \cdot \lambda) = (\mathrm{AB}) \cdot \lambda \tag{5}$$

The MatrixModifier component example is used to visualize the optimization process. At first the MatrixModifier model that is used to generate C/C++ code is analyzed. It has five input ports, one output port and consists of four subcomponents. The four subcomponents are one instance of a component called ModAdd, and three instances of a component called ModMul. Looking at the implementation math section of these components, and considering the given input types the number of operations can be estimated by our algorithm. As the ModAdd component instance takes two

1000x200 matrices as input and performs matrix addition the number of operations is 400,000 , as each element of mat1 is added to the corresponding element of mat2 and then stored which is also a 1000x200 matrix. Doing this for the other component instances results in a total of 20,700,000 operations for the MatrixModifier component. As this is a big number of operations, the generator collects all instructions from the subcomponents of MatrixModifier in the MatrixModifier component and analysis possible performance benefits from changing the order of operations. This results in the following cumulative matrix operation: As matrix multiplication is associative the execution order of multiplication operations can be changed. This is done by changing the operations in the following ways: Starting from the most left the different parts of the equation are evaluated by looking at the amount of steps required to perform the current mathematical operation. The equation is modified if changing it according to different rules results in a reduction of the total number of operation. For simplicity we only consider rules which can be derived from equations (2)-(5) here.
When looking at the MatrixModifier component the following equation can be extracted:

$$((\underset{1000\times 200}{\mathrm{A}} + \underset{1000\times 200}{\mathrm{B}}) \cdot (\underset{200\times 10}{\mathrm{C}} \cdot \underset{10\times 100}{\mathrm{D}})) \cdot \lambda \tag{6}$$

with $\mathrm{A} = \mathrm{mat1}$ , $\mathrm{B} = \mathrm{mat2}$, $\mathrm{C} = \mathrm{mat3}$, $\mathrm{D} = \mathrm{mat4}$ and $\lambda = \mathrm{factor}$. Starting on the outer left the first statement which is evaluated is the addition of A and B. This is then multiplied by $\mathrm{C}\cdot\mathrm{D}$. This case matches equation (2) and (4). However, if we would rewrite it to

$$(A \cdot (C \cdot D) + B \cdot (C \cdot D)) \tag{7}$$

, the amount of operations would increase, as now 3 multiplications (assuming that $\mathrm{C}\cdot\mathrm{D}$ is only calculated once) and 1 addition, instead of 2 multiplications and 1 addition are required, so it is not rewritten based on equation (2). According to equation (4) it can be rewritten as:

$$((\mathrm{A}+\mathrm{B}) \cdot \mathrm{C}) \cdot \mathrm{D}) \tag{8}$$

To decide whether the equation should be rewritten or not, the actual amount of operations done for these two cases have to be compared. In both cases a matrix addition is performed which results in a $1000 \times 200$ matrix, which amounts to $2 \cdot 1000 \cdot 200 = 400,000$ operations. When not changing the equation a matrix multiplication involving a $200 \times 10$ and $10 \times 100$ matrix is done, which results in $2 \cdot 200 \cdot 10 \cdot 100 = 400,000$ operations. However, when rewriting the equation a $1000 \times 200$ and a $200 \times 10$ matrix are multiplied. This requires $2 \cdot 1000 \cdot 200 \cdot 10 = 4,000,000$ operations. Lastly, in the starting equation a $1000 \times 200$ and $200 \times 100$ matrix are multiplied, which needs $2 \cdot 1000 \cdot 200 \cdot 100 = 40,000,000$ operations. In the rewritten equation a $1000 \times 10$ and $10 \times 100$ matrix are multiplied, this results in $2 \cdot 1000 \cdot 10 \cdot 100 = 2,000,000$ operations. So the rewritten equation takes a total of $6,400,000$ operations, and using the none rewritten equation costs $40,800,000$ operations. Therefore, a rewrite according to equation (4) should be performed. This results in:

$$(((\underset{1000\times 200}{\mathrm{A}} + \underset{1000\times 200}{\mathrm{B}}) \cdot \underset{200\times 10}{\mathrm{C}}) \cdot \underset{10\times 100}{\mathrm{D}}) \cdot \lambda \tag{9}$$

Now the evaluation continues with the rewritten equation. The next matched case is equation (5). The current equation performs a scalar multiplication on a $1000 \times 100$ matrix, which takes $1000 \cdot 100 = 100,000$ operations. When applying equation (5), a $10 \times 100$ matrix is used, which results in $10 \cdot 100 = 1,000$ operations. Therefore, the equation is rewritten. This results in the following equation:

$$((\underset{1000\times200}{A} + \underset{1000\times200}{B}) \cdot \underset{200\times10}{C}) \cdot (\underset{10\times100}{D} \cdot \lambda) \tag{10}$$

So, when not applying our optimizations the operations required are 40,900,000 and when applying our optimizations in this case results in only needing a total of 6,401,000 operations. Real world performance benefits can be seen in the case study in section 6.

### 4.2 Threading Optimizations

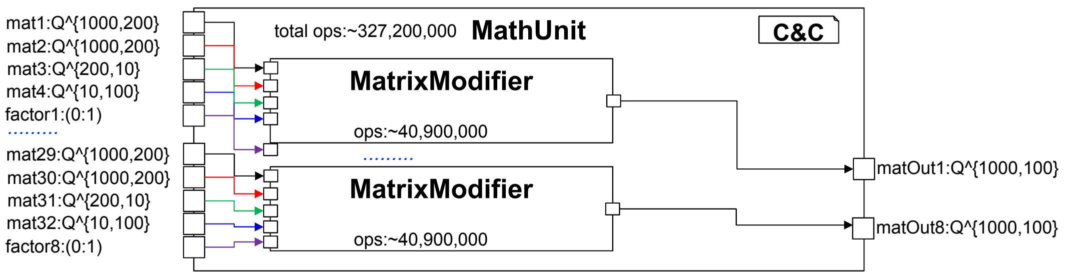

**Figure 4** C&C architecture of a MathUnit component which utilizes the MatrixModifier(figure 9) and does common matrix calculations, which are parallelizable.

We analyze the advantages and disadvantages of using multi-threading.
Threading optimizations are only needed when dealing with a multicore processor. Most modern CPUs consist of at least two cores, while those developed by Intel usually support hyperthreading(TM)[11], which enables one core to execute two threads. This is not as fast, as if there were two cores, each working on one thread, but hyperthreading does not suffer from issues like cache misses, regarding data that both threads need. This issue is obviously not present when both threads are executed on the same core, as they share the same cache. However, this can also cause problems if the cache size is really small, when both threads are only working with different data, as the amount of cache misses is increased for that case. So, to determine whether an application should utilize more threads, the best approach is testing. However, parallelization in general is only possible if different parts of the software are parallelizable. As threading in introduces a synchronization overhead, the generator has to be told manually what components should be threaded. Note that as the generator generates C++ code, existing tools like g++ support parallel compilation. Therefore, the generated code can also be compiled in parallel. MontiCore does not fully support parallelized usage yet. For this reason, parallel generation was not considered further and will be part of future work.

## 5 INCREMENTAL GENERATION

In this section a general approach of generating C/C++ code incrementally is examined.
As C++ code has increasingly long compilation times which are increased based on the amount source code to compile, the feature of incremental compilation and techniques for reducing the total compile time of each file are necessary. This is really important for increasing development speed, as, as far as we know, already existing software which is used heavily in the industry like Simulink [18], recompiles everything if anything is changed by default.
To utilize the already existing feature of incremental compilation support for C++-compilers like g++ from the GNU Compiler Collection[24], it is mandatory to generate the source code in a well structured and deterministic manner, so it becomes possible to only change those files which are affected by the changes that are done to the original model.
In general, that means our algorithm creates a function $\varphi : Components \rightarrow Files$ which maps a component $c$ to its corresponding file $f$. $\varphi$ is evaluated for every component that the generator has to generate C++ code for. As models which include themselves are not well defined, the generator can generate the C++ code of subcomponents always first. Note that if a component is not newly generated, the function will map the component to an empty file with not name, which means that the file for the component has not to be generated. Now we take a look at the different cases for components $c$ that are passed to $\varphi(c)$ and what the resulting files are:

- $c$ is a component which does have no math section and no subcomponents: This case is trivial as without any further changes the component will be generated as a file which corresponds to the name of the component if it was modified.
- $c$ is a component which has subcomponents(Note that a component may only contain a MontiMath behaviour description if it has no subcomponents): This file is newly generated if the component was modified or threading optimizations are toggled for this component.
- $c$ is a component that contains a math section: The file will be generated if the corresponding component was changed. However, as this component contains a MontiMath behaviour description, it is also re-generated if the used optimizations of the generator have changed since it was generated at first.

So the algorithm can be described in the following way and starts with the main component as argument: The component which is given as argument is checked whether the resulting file of $\varphi(c)$ already exists, if it does not, it is checked for changes. Then, the subcomponent will be evaluated by $\varphi$ and the algorithm is repeated for this component, until an atomic component, a component without subcomponents, is reached. As, components are not allowed to include themselves per definition this algorithm can not be stuck in a cycle and no infinite loop will occur.

## 6 CASE STUDY

The case study is done on a computer with an Intel(R) Core(TM) i7-6700HQ CPU @2.6 GHZ with 16GB DDR4 RAM clocked at 2133MHZ. The processor consists of 4 physical cores with Hyperthreading(TM) support, this makes for a total of 8 threads that can be executed at the same time.
The performance difference of two examples are studied when using the different optimizations. The optimization levels will be explained in the following:

- Optimization Level 0 (L0): No optimizations are applied.
- Optimization Level 1 (L1): Threading optimizations are applied.
- Optimization Level 2 (L2): All Algebraic optimizations are applied.
- Optimization Level 3 (L3): Algebraic and Threading optimizations(L1 and L2) are applied.

Note that the code was compiled using the -03 flag.

The first example shows the execution time of the `MatrixModifier` example which is shown in figure 9. To underline the measurable performance gains which are received from the applied optimizations, the amount of `MatrixModifier` components inside of the `MathUnit`(shown in figure 4) component are different. In table 1 `MatrixModifier` is used as a reference to better compare the execution times of `MathUnit`. The different amount of `MatrixModifier` subcomponents results in better comparability. The measurements for the `ObjectDetector` component can be seen similar to this in table 2.
When looking at the percentage values it can be seen that the algebraic optimizations always result in a performance gain. However, threading overhead causes slower execution times when the amount of threads does not correspond to the amount of cores. This also includes the usage of threads, when the program could be executed in the main thread. In the `ObjectDetector` comparison, peak performance (46.65%) is achieved when using 4 threads which is to be expected as the used CPU has 4 physical cores. However, for the `MathUnit` this is not the case. When using 6 threads, the performance was slightly better as it was 61.03% while it was 63.41% when using 4 threads. This could be related to the way the CPU prefetches instructions in this case. In the real world example, execution times could be reduced by 53.35In the worst case execution took 5.96% longer, which was caused by the threading overhead. Therefore, it is important to only use the threading optimization when a component is parallelizable and the CPU has appropriate threading capabilities.

## 7 IMPLEMENTATION

After explaining the general concepts of our generator, a short description of the implementation is given in the following. The generator is implemented in Java, as the whole MontiCore[12] toolchain is also written in Java. This allows the utilization of all features of MontiCore, like the generated abstract syntax tree classes and created symbols inside of the symbol table, for more detail see [13]. In general the generator receives information about each component, that exists in a directory. Then the generator examines all parts of each component and performs different actions when different information is encountered. This includes information like, does a component have an *implementation Math* section, but also information which is related to the execution order. It computes the execution order based on the following rules:

- Components that have no input ports are executed before components that contain input ports.
- A component `A`, which has input ports that are connected to the output ports of a component `B` must always be executed after the component to which it is connected. This means that component B is executed before component A.

These rules are similar to the execution ordering rules of models that were developed and executed using Simulink. Therefore, models that are converted from Simulink to MontiCAR should be executed in the same order.
As MontiCAR models may contain an *implementation Math* section in which the behaviour is described by using mathematical expression, there are various classes that are derived from a `MathExpressionSymbol` class. The conversion of the mathematical expressions to C++ code is done by using a recursive algorithm that iterates through every mathematical expression. This is needed, as expressions may contain complex subexpression. One example is a multiplication expression. It contains two subexpressions which might be expressions that consist of subexpressions themselves. The actual implementation of the generator and the MontiCAR language family is available at `https://github.com/EmbeddedMontiArc`.

## 8 RELATED WORK

When examining C&C modeling, usually the creation and analysis of models is prioritized [3, 7]. The definition of a domain-specific modeling language and how it is transformed to another language can be done by using MetaEdit+[25]. Other software which allows the creation of domain-specific languages and further tooling includes the MontiCore language workbench[12] and Xtext[6]. The composition of already existent code generators is described in [22]. As the MontiCAR language family is relatively new, no code generators for this language were created previously. Therefore, there exists no further directly related work that discusses the development of a code generator for the MontiCAR language family. However, an explanation for a framework which deals with the modeling, simulation and code generation of sensor network applications can be read in detail at [20]. In this paper the domain-specific knowledge related to network sensors is used and a short introduction to the used technologies is given. This includes Simulink, TinyOS[17] and MANTIS[2]. A different approach for modeling and code generation in a web environment, which is done on the documentation level, is presented in [14]. They created a tool called MOSAIC that is used to model and simulate chemical processes [9].

## 9 CONCLUSION

We presented a method to develop a generator which can be used to generate executable code out of a C&C model described by utilizing the MontiCar language. In contrast to other existing generators for tools like MatLab/Simulink, we automatically use incremental generation and generate multithreaded code when performance can be improved by parallelizing the execution of different parts of a C&C model at the same time, while also rewriting algebraic operations if this rewrite results in faster code. In future work, the automatic generation of code that is executed by the GPU, as a possibility to further reduce the time to complete complex computations has to be examined. The generation of parallel code is currently only done automatically for the main component of a model, as a complex analysis of the complete model is needed to determine a new execution order which does not violate execution order rules but also uses the available threading capabilities of a CPU as optimal as possible. However, big models usually contain different submodels that are independent from each other. The evaluation in section 6 also showed, that by using threads which are reasonable in relation to the amount of cores of a CPU, a performance benefit is gained from parallelization. The development of a better algorithm will also be part of future work.

## References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[2] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. “MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms”. In: *Mobile Networks and Applications* 10.4 (2005), pages 563–579.

[3] Manfred Broy, Franz Huber, and Bernhard Schätz. “AutoF ocus–Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme”. In: *Informatik-Forschung und Entwicklung* 14.3 (1999), pages 121–134.

[4] Bernard Desgraupes. “Clustering indices”. In: *University of Paris Ouest-Lab Modal'X* 1 (2013), page 34.

[5] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computations*. 2016. URL: http://www.gnu.org/software/octave/doc/interpreter.

[6] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2010, pages 307–309.

[7] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[8] Franz E Hohn. *Elementary matrix algebra*. Courier Corporation, 2013.

[9] R Idris, CT Hing, N Harun, and MR Othman. "Development of Equation Oriented Modelling of Advanced Distillation Process Using MOSAIC: Dividing Wall Column Case study". In: (2017).

[10] MKL Intel. "Intel math kernel library". In: (2007).

[11] David Koufaty and Deborah T Marr. "Hyperthreading technology in the netburst microarchitecture". In: *IEEE Micro* 23.2 (2003), pages 56–65.

[12] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: a Framework for Compositional Development of Domain Specific Languages". In: *International Journal on Software Tools for Technology Transfer (STTT)*. Volume 12. 2010, pages 353–372.

[13] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: a framework for compositional development of domain specific languages". In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.5 (2010), pages 353–372.

[14] Stefan Kuntsche, Tilman Barz, Robert Kraus, Harvey Arellano-Garcia, and Günter Wozny. "MOSAIC a web-based modeling environment for code generation". In: *Computers & Chemical Engineering* 35.11 (2011), pages 2257–2273.

[15] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. "Modeling Architectures of Cyber-Physical Systems". In: *European Conference on Modelling Foundations and Applications*. Springer. 2017, pages 34–50.

[16] Edward A Lee. "Cyber physical systems: Design challenges". In: *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on*. IEEE. 2008, pages 363–369.

[17] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. "TinyOS: An operating system for sensor networks". In: *Ambient intelligence* 35 (2005), pages 115–148.

[18] Mathworks. *Simulink User's Guide*. Technical report R2016b. MATLAB & SIMULINK, 2016, page 4022.

[19] MATLAB. *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[20] Mohammad Mostafizur Rahman Mozumdar, Francesco Gregoretti, Luciano Lavagno, Laura Vanzago, and Stefano Olivieri. "A framework for modeling, simulation and automatic code generation of sensor network application". In: *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on*. IEEE. 2008, pages 515–522.

[21] Andrew Y Ng, Michael I Jordan, and Yair Weiss. “On spectral clustering: Analysis and an algorithm”. In: *Advances in neural information processing systems*. 2002, pages 849–856.

[22] Jan Oliver Ringert, Roth Alexander, Rumpe Bernhard, and Wortmann Andreas. “Language and code generator composition for model-driven engineering of robotics component & connector systems”. In: *JOURNAL OF SOFTWARE ENGINEERING IN ROBOTICS* 6.1 (2015), pages 33–57.

[23] Conrad Sanderson and Ryan Curtin. “Armadillo: a template-based C++ library for linear algebra”. In: *Journal of Open Source Software* (2016).

[24] Richard M Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3. 3*. CreateSpace, 2009.

[25] Juha-Pekka Tolvanen and Matti Rossi. “MetaEdit+: defining and using domain-specific modeling languages and code generators”. In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM. 2003, pages 92–93.

[26] Ulrike Von Luxburg. “A tutorial on spectral clustering”. In: *Statistics and computing* 17.4 (2007), pages 395–416.

[27] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. “Intel math kernel library”. In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pages 167–188.

[28] David S Watkins. *Fundamentals of matrix computations*. Volume 64. John Wiley & Sons, 2004.

[29] Zhang Xianyi, Wang Qian, and Werner Saar. “OpenBLAS: An optimized BLAS library”. In: *Accedido: Agosto* (2016).

## APPENDIX

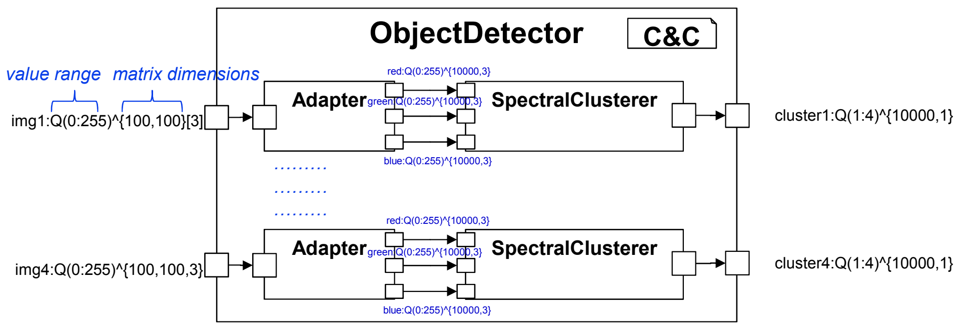

**Figure 5** C&C architecture of the ObjectDetector.

EMA

```
component ObjectDetector {
 ports in Q(0:255)^{2500,3} imgFront,
       in Q(0:255)^{2500,3} imgBack,
       in Q(0:255)^{2500,3} imgLeft,
       in Q(0:255)^{2500,3} imgRight,
       out Q(-oo:oo)^{2500,1} clusters[4];

 instance SpectralClusterer<2500,4,4> spectralClusterer[4];

 connect imgFront -> spectralClusterer[1].imgMatrix;
 connect imgRight -> spectralClusterer[2].imgMatrix;
 connect imgLeft -> spectralClusterer[3].imgMatrix;
 connect imgBack -> spectralClusterer[4].imgMatrix;

}
```

**Figure 6** ObjectDetector model defined in MontiCar.

```
component KMeansClustering<N1 n, N1 amountVectors, N1 maximumClusters> {
 ports in Q(-oo:oo)^{n, amountVectors} vectors,
       out Q(-oo:oo)^{n, maximumClusters} clusters;

 implementation Math{
   Q^{n,amountVectors} UMatrix;   Declare matrix UMatrix
   for i=1:size(vectors,1)   For loop from 1 until size of first dimension of vectors
     Q^{1,amountVectors} target= vectors(i,:) .^ 2;  Set target vector to squared i'th vector of vectors
     Q amount = sqrt(sum(target));     Set amount to square root of the sum of all entries in target
     UMatrix(i,:) = vectors(i,:) ./ amount;  Set i'th column of UMatrix to normalized element of vectors
 end

 clusters = kmeans(UMatrix, maximumClusters);Set output port clusters to result of kmeans algorithm
}}
```

**Figure 7** KMeansClustering model defined in MontiCar.

SubComponent of SpectralClusterer

EMA

```
component NormalizedLaplacian<N1 n> {
 ports in diag Q(-oo:oo)^{n,n} degree,
       in Q(-oo:oo)^{n,n} W,
       out Q(-oo:oo)^{n,n} nLaplacian;

 implementation Math{
   nLaplacian = degree^(-0.5) * W * degree^(-0.5);

}}
```

Could also be replaced by:
```
for i=1:size(degree,1)
  for j=1(degree,2)
    nLaplacian(i,j) = W(i,j)/
    (sqrt(degree(i,i)*sqrt(degree(j,j));
  end
end
```

**Figure 8** NormalizedLaplacian model defined in MontiCar.

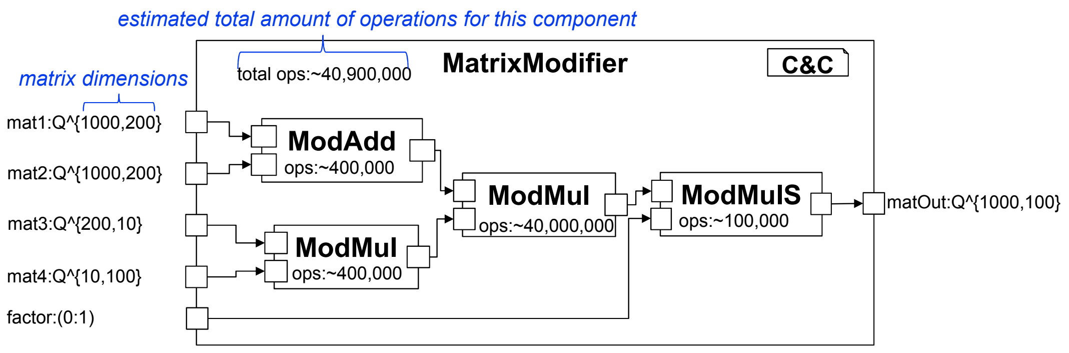

**Figure 9** C&C architecture of a component which does common matrix calculations.

**Table 1** Comparison of the time the generated code took to execute 100 cycles of the depicted `MathUnit` instances. The number of `MatrixModifier` components used by a `MathUnit` instance is shown in brackets. The measurements are shown as absolute values, and as percentage values, with the execution time of the corresponding `MathUnit` instance when no optimizations are applied.

| Example | Optimization Level 0 (L0) | Optimization Level 1 (L1) | Optimization Level 2 (L2) | Optimization Level 3 (L3) |
|---|---|---|---|---|
| MathUnit (1 MatrixModifier) | 164 ms | 141 ms | 188 ms | 169 ms |
| MathUnit (2 MatrixModifier) | 295 ms | 274 ms | 268 ms | 220 ms |
| MathUnit (3 MatrixModifier) | 420 ms | 372 ms | 359 ms | 297 ms |
| MathUnit (4 MatrixModifier) | 552 ms | 485 ms | 462 ms | 350 ms |
| MathUnit (5 MatrixModifier) | 709 ms | 676 ms | 579 ms | 443 ms |
| MathUnit (6 MatrixModifier) | 857 ms | 764 ms | 677 ms | 523 ms |
| MathUnit (7 MatrixModifier) | 935 ms | 876 ms | 812 ms | 622 ms |
| MathUnit (8 MatrixModifier) | 1113 ms | 978 ms | 900 ms | 750 ms |
| MathUnit (9 MatrixModifier) | 1184 ms | 1091 ms | 1024 ms | 859 ms |
| MathUnit (1 MatrixModifier) | 100% | 85,98% | 114,63% | 103,05% |
| MathUnit (2 MatrixModifier) | 100% | 92,88% | 90,85% | 74,58% |
| MathUnit (3 MatrixModifier) | 100% | 88,57% | 85,48% | 70,71% |
| MathUnit (4 MatrixModifier) | 100% | 87,86% | 83,70% | 63,41% |
| MathUnit (5 MatrixModifier) | 100% | 95,35% | 81,66% | 63,48% |
| MathUnit (6 MatrixModifier) | 100% | 89,14% | 79,00% | 61,03% |
| MathUnit (7 MatrixModifier) | 100% | 93,69% | 86,84% | 66,52% |
| MathUnit (8 MatrixModifier) | 100% | 87,87% | 80,86% | 67,39% |
| MathUnit (9 MatrixModifier) | 100% | 92,15% | 86,49% | 72,55% |

**Table 2** Comparison of the time the generated code took to execute 1 cycle of the depicted `ObjectDetector` instances. The number of `SpectralClusterer` components used by a `ObjectDetector` instance is shown in brackets. The measurements are shown as absolute values, and as percentage values, with the execution time of the corresponding `ObjectDetector` instance when no optimizations are applied.

| Example (amount of SpectralClusterer components) | Optimization Level 0 (L0) | Optimization Level 1 (L1) | Optimization Level 2 (L2) | Optimization Level 3 (L3) |
|---|---|---|---|---|
| ObjectDetector (1) | 35398ms | 21079ms | 36286ms | 37509ms |
| ObjectDetector (2) | 72272ms | 41044ms | 47372ms | 44549ms |
| ObjectDetector (3) | 106140ms | 62154ms | 60172ms | 52821ms |
| ObjectDetector (4) | 142917ms | 82413ms | 73458ms | 66672ms |
| ObjectDetector (5) | 178939ms | 105555ms | 92955ms | 88418ms |
| ObjectDetector (6) | 216404ms | 124998ms | 118830ms | 112331ms |
| ObjectDetector (7) | 251746ms | 144398ms | 146298ms | 124502ms |
| ObjectDetector (8) | 289919ms | 163262ms | 183209ms | 150237ms |
| ObjectDetector (9) | 322129ms | 180868ms | 213445ms | 192656ms |
| ObjectDetector (1) | 100% | 59.55% | 102.51% | 105.96% |
| ObjectDetector (2) | 100% | 56.79% | 65.55% | 61.64% |
| ObjectDetector (3) | 100% | 58.56% | 56.69% | 49.77% |
| ObjectDetector (4) | 100% | 57.66% | 51.40% | 46.65% |
| ObjectDetector (5) | 100% | 58.99% | 51.94% | 49.41% |
| ObjectDetector (6) | 100% | 57.76% | 54.91% | 51.91% |
| ObjectDetector (7) | 100% | 57.36% | 58.11% | 49.46% |
| ObjectDetector (8) | 100% | 56.31% | 63.19% | 51.82% |
| ObjectDetector (9) | 100% | 56.15% | 66.26% | 59.81% |