Conception and Implementation of Parallelism Analyses in MATLAB/SIMULINK Models for programming Embedded Multicore-Systems

Dominik Chessa
Konzeption und Implementierung von Parallelitäts-Analysen in MATLAB/SIMULINK Modellen für die Programmierung Eingebetteter Multicore-Systeme

Conception and Implementation of Parallelism Analyses in MATLAB/SIMULINK Models for programming Embedded Multicore-Systems

Author: Dominik Chessa
Supervisor: Prof. Dr. Dr. h.c. Manfred Broy
Advisor: Wolfgang Schwitzer
Submission Date: 15. Juni 2011
I assure the single handed composition of this bachelor thesis only supported by the declared resources.

Date Signature
Abstract

This paper addresses the conception and implementation of analyses in MATLAB/Simulink models, which are then utilized to evaluate a Simulink model’s concurrency for implementation on embedded multicore systems. For this purpose, Simulink models are translated into discrete synchronous dataflow graphs without compromising their structure, leaving subsystems as well as precedence relationships intact. Assumptions are made to ensure the concurrency of the resulting model as well as to ensure structural integrity.

A cost model is applied to the graph, enabling it to be scheduled using fully-static rate scheduling, minimizing its iteration period and resulting in metrics such as speedup and efficiency. Using these metrics, an industrial Simulink model is analyzed and a fitting architecture is proposed using several angles for interpretation such as technical and economical factors.
## Contents

1 Introduction .................................................. 1  
   1.1 Motivation .............................................. 1  
   1.2 Goal ..................................................... 3  
   1.3 Outline of this Thesis .................................... 5  

2 Related Work and Concepts ................................. 6  
   2.1 Simulink .................................................. 6  
      2.1.1 Simulink Blocks ..................................... 6  
      2.1.2 Simulink Model ...................................... 11  
      2.1.3 MDL File ............................................. 12  
   2.2 ConQAT .................................................... 13  
      2.2.1 Processors and Blocks ............................... 13  
      2.2.2 Simulink Parser ..................................... 14  
   2.3 Dataflow Graph ........................................... 15  
      2.3.1 Synchronous Dataflow Graph ......................... 17  
      2.3.2 Homogeneous Synchronous Dataflow Graph ............ 18  
   2.4 Allocation and Scheduling ............................... 20  
      2.4.1 Fundamentals of Scheduling ......................... 20  
      2.4.2 Fully Static Scheduling ............................. 21  
      2.4.3 Scheduling Homogeneous Synchronous Dataflow Graphs 22  
   2.5 Metrics for analyzing parallel Systems ................ 23  
      2.5.1 Makespan ........................................... 23  
      2.5.2 Speed-Up ........................................... 24  
      2.5.3 Efficiency ......................................... 24  
   2.6 Related Work ............................................. 25  

3 Implementation of the Simulink Translator ............... 27  
   3.1 General Concepts ......................................... 27  
      3.1.1 Assumptions for Concurrent Systems ................ 27  
      3.1.2 Translation Depth .................................... 28  
   3.2 Process of Translating Simulink Models ................. 29  
      3.2.1 Blocks to Vertices ................................... 29  
      3.2.2 Lines to Edges ....................................... 32  
      3.2.3 Creating the Graph .................................. 32  
      3.2.4 Adding Delays ....................................... 33  
      3.2.5 Unfolding the Graph ................................ 34  
   3.3 Translating Specific Block-Classes ..................... 35  
      3.3.1 Sources and Sinks ................................... 36  
      3.3.2 Function Blocks ...................................... 37
3.3.3 Delays / Discrete Data Storage 42
3.3.4 Structural Blocks 43
3.3.5 Signal Routing Blocks 45

4 ConQAT Integration 49
4.1 Why to use ConQAT? 49
4.2 The SimulinkDataflowTranslation Processor 49
4.3 The DataFlowScheduleProcessor 50

5 An Industrial Case-Example - The ACC System 51
5.1 Motivation 51
5.2 The Adaptive Cruise Control Simulink Model 53
5.3 Experiment Setup 55
5.4 Experiment Results 56
5.4.1 General Observations 57
5.4.2 Evaluation of the Results 60
5.4.3 Conclusion 62

6 Discussion and Future Work 63
6.1 Discussion 63
6.1.1 Cost Model 63
6.1.2 Assumptions for translating Simulink Models 64
6.2 Future Work 65
6.2.1 Vertex Functionality 65
6.2.2 Precise Complexity Analysis 67
6.3 Conclusion 68

A Appendix 76
1 Introduction

1.1 Motivation

Building efficient multicore-based Embedded Software Systems. Used in most every-day items today, Embedded Systems are one of the main focuses of the industry right now. Mobile phones, internet routers, microwaves, cars, planes, even washing machines all contain them. Small, contained systems designed to carry out one or more specific tasks. Their requirements are all the same: a low power consumption, small size and low heat generation, combined with functional safety, reliability as well as low cost. Since more and more functionality is to be implemented in these items, functional requirements for these systems also rise. More computational power is needed. One way to remedy this problem is to apply the principles of Moore’s Law and increase the number of transistors, thus trying to increase the frequency the processor runs on. This increases power consumption as well as heat generation, factors that need to be decreased when dealing with Embedded Systems. Other possible solutions deal with the same problems, such as distributed systems (connecting multiple single-core processors).

Reaching the limits of Moore’s Law. In 1965, Intel co-founder Gordon E. Moore published a paper in “Electronics”-magazine. The containing ideas and thoughts went on to be called “Moore’s Law”. While undergoing some changes and sometimes being defined differently, the essence of Moore’s Law is that transistors on electronic semiconductors double in specific time intervals - around 18 months. While this has been true until now, Moore’s Law might hit the so-called Power Wall sooner or later. Not only will it someday not be possible to produce even smaller transistors, it will also become more and more costly over time, resulting in the need to evaluate cost and benefit. Increasing the transistors on a chip, as already mentioned, results in higher electric power consumption as well as rising heat emission, making the processor less suited to be used in low-power systems such as Embedded Systems.
Multi-Core architectures. A promising solution to this problem, and a way to further double the number of transistors, thus sustaining Moore’s Law, is to switch from single to multi-core architectures as power consumption on multi-core processors scales better than when utilizing multiple single-cores. Having a quadcore processor consumes less electrical power than using one single-core processor, having four times the frequency in order to achieve the same computing power.

Due to this, the industry is slowly but surely switching over to multi-core architectures, forcing companies to start developing with parallelism in mind.

Model-based approach. Developing parallel systems calls for different methods in order to ensure a high degree of concurrency. According to *Amdahl’s Law*, the speedup of a program running on a multi-core is limited by the sequential portion of the program [Amd67]. By using a model-based approach to development of an application’s implementation, the system can be analyzed and put together more efficiently, keeping the sequential part of the program comparably small. Since change occurring late in a design process is more expensive than early on [Boe81], this approach also prevents unnecessary expenses.

Modeling the task with tools like Simulink can give a clear overview of the system as well as a common vocabulary. It also makes it easier to decide on requirements the system will need to satisfy. Using that model it is possible to automatically analyze a model for various aspects such as how well a task can be parallelized and run on multiple cores. These analyses can be run without human interaction with tools such as ConQAT, delivering a nightly or weekly build which tells developers how well they meet the expected quality.

Matlab/Simulink. One tool for modeling reactive systems is Simulink by MathWorks. Since according to [HM09], 58% of the respondents use Simulink for Embedded Systems modeling, followed by Scade with 25%, it can be considered one of the most wide-spread software solutions for modeling real-time embedded systems. This makes it an obvious choice as the focus of this work and the starting position for analyses made.
The problem is, that modeling parallel systems in tools such as Simulink is possible, but as of today, architectures are optimized to run on multiple cores. With an adequate set of analyses, a system can be fitted on a multi-core architecture while ensuring compatibility and a high degree of efficiency.

**Economical view.** Since Embedded Systems are supposed to carry out very specific tasks, they can be optimized for process efficiency as well as cost-effectiveness. Depending on the task at hand, various kinds of processors with different prices can be used. Considering that components are mostly bought in bulk, even a small cost reduction can make a big difference. For example, if a company bought thousands of eight-core processors to run an application that only utilizes 4 of those, half of the capacity is not used. This would result in an inefficient system - not only technically but also financially, since buying a cheaper processor with 4 cores would have resulted in the same performance for less cost. Thus, using a model-based approach to determine cost as a requirement can lead to a financial gain.

### 1.2 Goal

The goal of this thesis is the inspection of concepts for parallelism analyses of MATLAB/Simulink models. Developed analysis techniques are then implemented in the continuous software quality analysis tool ConQAT and tested on an industrial MATLAB/Simulink model.

The following five points are defined as this thesis’ main goals:

- Making presumptions that need to be fulfilled to be able to analyze parallelism in MATLAB/Simulink.
- Discussion of basic Data Flow methods.
- Defining how to gather conclusions concerning concurrency of the analyzed model from the results.
- Implementation of selected analysis methods as processors in ConQAT.
• Evaluation of the acquired concepts by executing them on an industrial MATLAB/Simulink model.

In the course of this thesis, our analyses are evaluated through four dimensions as seen in Figure 1.

![Four Dimensions of this Thesis](image)

Figure 1: Four Dimensions of this Thesis

This leads to one other goal of this thesis. To make conclusions on how the number of utilized cores, the architectural granularity and the message delay/communication time affect parallelism of a Simulink model. We also view concurrency of a system as an economic/financial factor and conclude the degree in which these decisions can impact financial gain.
1.3 Outline of this Thesis

The points stated in Section 1.2 are organized as follows. In Section 2, general concepts used in this thesis are explained, starting off by explaining the essential functions and structure of MATLAB/Simulink models, followed by ConQAT and concepts such as dataflow graphs, allocation, scheduling as well as metrics used in this work. The rest of the section presents a related work, also dealing with translating MATLAB/Simulink models into a synchronous dataflow language.

Sections 3 and 4 deal with the translation of Simulink models into dataflow graphs, which was developed in the course of this thesis, and its integration in ConQAT. After making presumptions needed for concurrent systems, each step of the translation algorithm is explained, followed by detailed descriptions of the assumptions we made for the most used Simulink blocks. In Section 4, reasons for the use of ConQAT are given before explaining the processors that have been implemented for our analysis.

In Section 5, these analyses are applied on part of an industrial Simulink model, the used hardware as well as software setup is specified, the results that are received are described and interpreted with several metrics and views. Finally Section 6 discusses all previous topics, unfinished parts such as the functional translation are detailed with thoughts on how they can be implemented in the future and a conclusion is given.
2 Related Work and Concepts

The following sections aims to explain the fundamental concepts used in this thesis such as dataflow graphs, MATLAB/Simulink and scheduling of tasks to resources, as well as show the extent of how they are used.

2.1 Simulink

“Simulink is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems” [Mat11a]. It provides multiple blocks that can be dragged around a workspace and connected through ports with lines - establishing an input output relationship as well as dependencies between those two blocks.

The following section describes the main components of a Simulink model and aims to give an overview of how Simulink works. Please note that not all of the functionality in this section is relevant to the translation. The sample time, for example, is always presumed to be discrete. For more details check the Simulink reference at [Mat11b].

2.1.1 Simulink Blocks

Atomic blocks. In Figure 2, we see a typical Simulink block with it’s imports and outports, depicted by triangles either pointing to or away from the block. Imports dictate how many signals a block can receive, these signals are then processed by the block and output through the outports. It is also possible for a block to have no imports (source blocks), no outports (sink blocks) or both (i.e. model-wide utilities). What happens to the received
signals depends on the type of the block. This particular block takes the input from its two inports and outputs the sum through its outport.

Hierarchical in Simulink. It is to be noted that a block can also contain another Simulink model, thus making it a SubSystem. It is used just like a regular block, with its inports linked to specific inport-blocks.

Block classes. Simulink contains different kinds of blocks: continuous blocks, discrete blocks, logic operations, math operations, routing blocks, sinks, sources and more. For the sake of simplicity we can divide Simulink blocks into the following classes:

- Input/Output Blocks (or “Sources and Sinks”)
- Function Blocks
- Delays / Discrete Data Storage
- Structural Blocks
- Signal Routing Blocks

In this section, we examine these classes and look at a few blocks from each one in further detail.

Sources and Sinks. While most blocks contain at least one input- and one output-port, sources and sinks are different in that regard. A source does not have any input ports, it only sends data to any block connected to it. This can be either input from the environment or block-generated data like constants, sine waves and more.

Sinks on the other hand do not have an output port. Every signal they receive ends there. This can be used to output data to the environment, show numerical values on the display, to discard data or to completely end
Function Blocks  As already stated, most blocks receive signals through an import, do something with that signal and output the result through an outport. In this work, most blocks like this are categorized as Function Blocks. Most function blocks are mathematical operations. These blocks
provide functionality in the vein of addition, subtraction, division, multiplication, dot product, matrix and vector concatenation and more.

![Mathematical Operation Blocks](image)

Figure 4: Mathematical Operation Blocks

Figure 4 shows four of these blocks. *Gain* multiplies the incoming signal with a constant factor *Product* and *Add* multiply or add two signals together while *MinMax* takes a vector and outputs the minimum or maximum value, depending on which variable is set.

Other function blocks include integrators, rate limiters, logic and bit operations as well as user-defined MATLAB functions.

**Delays / Discrete Data Storage** Delays are blocks that hold the input for a specified sample time.

![Delay Blocks](image)

Figure 5: Delay Blocks

Figure 5 shows the three most used delay blocks. The *UnitDelay* block both accepts and outputs signals with a discrete sample time and delays them for a specified period. The *Memory* block accepts and outputs signals with a continuous or fixed sample time and delays them for one time step. The *Zero Order Hold* works like a UnitDelay apart from the ability to accept signals with continuous sample time and output a signal with discrete sample time.
For this work we assume that every block works with a discrete sample time. That means that if we see a Memory block, we assume it works the same as a UnitDelay block with a specified sample time of 1. For further details on this, see Section 3.3.3.

**Structural Blocks** As already stated a Simulink block can also contain other blocks. This block is then called a SubSystem block. Each port on this block corresponds with either an InPort-Block or an OutPort-Block, which forward the data given to them to one or more blocks within the SubSystem. Thus the block can be used to give the user a clearer overview over the system. This can both be cosmetic, for example by assigning blocks to SubSystems so the model can be easier read by the human eye, or functional, by using SubSystems to show borders between real components.

![Figure 6: Structural Blocks](image)

Figure 6 shows the three most important structural blocks. Just as most structural blocks, Enabled and Triggered SubSystems are special SubSystems.

*Enabled SubSystems* contain a EnablePort block which acts like an InPort. If the SubSystem receives a positive signal on this port it is activated. Thus, an Enabled SubSystem is used when it is necessary to activate a SubSystem at a specific time. It can either be set to “held” or “reset” which signifies if the value is reset every time a signal is received or if the last one is held.

*Triggered SubSystems* resemble Enabled Subsystems in that they contain a special port, the TriggerPort. It can be set to different states that define
the value a signal should contain so the SubSystem will be active: *rising, falling, either or function call.*

**Signal Routing Blocks** Simulink models use signals as inputs and outputs of each block. Signal routing blocks can be used to change these signals.

![Diagram of signal routing blocks](image)

**Figure 7: Signal Routing Blocks**

Figure 7 shows a selection of blocks that work with signals. The *BusCreator* and *BusSelector* pair are used to bundle a set of signals into a bus and output specified signals from that bus again. The *Mux* and *Demux* pair works in a similar way: several signals get combined into a vector, which can be extracted again when they are needed. The difference between bussed and muxed vectors is that the BusSelector can specify which signals to extract while the demux blocks just splits up the vector into as the same number of signals there were before.

The *From* and *Goto* blocks are used to route a signal to different parts of the model. Each Goto block passes its input to one or more corresponding From blocks while each From block receives its output from only one specified Goto block. This way a signal can be transported through several layers.

### 2.1.2 Simulink Model

A model is created by connecting various blocks through directed lines pointing from a block’s outport to another block’s inport.

Figure 8 shows a Simulink model that calculates \( z = \frac{10(x+y)}{2} \), with \( x \) and \( y \) being the the values from the inports and \( z \) being the result given to the outport. \( z \) is them output to the environment (through the outport) while
It is also possible to divide a line into two which is called a branch line (see the line after the Product block). This way an outport can be connected to multiple inports which is not possible the other way around as each inport can only receive one signal. It is also not allowed to connect a block to itself. If a self-loop is desired, a delay-block can be used to save the value for one iteration period.

2.1.3 MDL File

Simulink models can be saved as .mdl-files which is an ASCII file containing information describing the model. Each section is started by a keyword filled with pairs of parameters and values.

It contains information about each system and subsystem and their contained blocks, ports and lines in hierarchical order. It also consists of sections describing global variables. These are too numerous to list here. A list can be found in [Mat11b].
2.2 ConQAT

Whether it is productive to use a specific software product is often influenced by the quality of the software. To neglect quality aspects such as maintainability leads to greater costs in the long run [BBL76]. To remedy this, the system should be analyzed in regular intervals to ensure a specified level of quality. Conducting the analysis manually is not possible anymore due to the size of recent software development projects, thus a tool has to be used for the process. The “Continuous Quality Assessment Toolkit” ConQAT provides tools to monitor and analyze software projects for their quality [DS06].

It monitors a system’s state by continuously assessing it and visualizing the data in a comprehensive way, allowing developers to gain an overview over their system’s quality as well as enabling them to conduct in-depth analyses of a part of the system. These analyses can be specified and adapted by the developers, thus tailoring it to the project. These three activities (Monitoring, In-Depth Analysis, Tailoring) make up the Core Activities of continuous quality control [FDLHJ10].

This section explains the aspects of ConQAT used in this work.

2.2.1 Processors and Blocks

Similar to Simulink blocks, ConQAT processors are the basic building blocks of a ConQAT analysis, connected by lines. Each processor implements a specific function used in the analysis by taking a specified number of inputs and generating one output signal. These typed inputs and outputs are called parameters. The results are then carried via edges as Transport Objects. It can be said that, structurally, a ConQAT processor is very similar to a Simulink block. Thus an analysis, just like a Simulink model, can be visualized as a data flow graph, as seen in Figure 9.

ConQAT also implements Blocks, that contain a series of connected pro-
cessors to prevent repeated elements. A ConQAT block is similar to a Simulink subblock in the regard that it contains atomic building blocks as well as “input” and “output” specifiers.

In this work, we implement various processors, carrying out a Simulink model analysis. The process is detailed in Section 4.

2.2.2 Simulink Parser

ConQAT provides us with a parser that can extract information from MDL-Files. Through it we can access model parameters as well as every simulink element. The parser is seperately available as an open-source library from [Web11a].

After it is given a file it translates it into a SimulinkModel object which contains every parameter and its corresponding value that is noted in the model file. It also generates objects representing each structural component contained in the model: SimulinkBlocks, SimulinkLines, SimulinkInPorts and SimulinkOutPorts. A SimulinkBlock object can in turn contain more SimulinkBlocks, lines and ports, making it a subsystem.
Figure 10: UML Diagram of the Simulink Model representation (Stateflow not included)

Figure 10 shows the relationship between the objects as well as their abstract representation. The missing part involves Stateflow, which is not a part of this thesis. The parser also creates SimulinkAnnotation objects, which are basically comments in Simulink models, but in this work we only access blocks, lines and ports. More information about the parser - such as Stateflow integration and even source code - can be obtained from the API documentation at [Web11b].

2.3 Dataflow Graph

A data flow is a programming model in which a program is represented as a set of tasks with data precedences. It is represented by a directed multigraph (a generalization of a directed graph), an ordered pair of vertices and edges $G = (V, E)$, with each edge being an ordered pair $e = (v_i, v_j)$ where $v_i, v_j \in V$. In $e = (v_i, v_j) \in E$, a directed edge connecting the vertices $v_i$ and $v_j$, $v_i$ is
called the source vertex of $e$, while $v_j$ is called the sink vertex of $e$ ($src(e)$ and $snk(e)$ in short). In a directed multigraph, more than one edge can have the same source and sink vertex [SB09].

![Dataflow Graph]

Figure 11: A dataflow graph with four vertices and three edges.

In a data flow, actors (vertices) consume and/or produce so-called tokens. Once all needed inputs are available, an actor can fire (start its computation). In Figure 11 we can see a typical dataflow graph. Actors A and B do not have any input arcs and can fire at any time, making them possible data sources. D on the other hand does not send any tokens, making it a data sink. The edges not only define which actors send data to each other but also show a precedence relationship. In this example, A and B can fire simultaneously, resulting in a concurrent part of the model. In order for C to fire, both A and B need to be finished producing their outputs and only after C has finished its computation, D can fire.

In this work, we use a Homogenous Synchronous Dataflow Graph, a special case of a Synchronous Dataflow Graph as our Model of Computation (MoC) during Simulink translation. An MoC defines the operations allowed in a computation as well as their costs, thus providing semantics to use in a computational structure [EJL+03]. This section explains both of these dataflow types.
2.3.1 Synchronous Dataflow Graph

The Synchronous Data Flow is a model proposed by Lee and Messerschmidt in 1987 [LM87]. A synchronous data flow graph (SDF graph) limits the number of tokens produced or consumed by an actor to a fixed value that is specified beforehand. This number of tokens consumed and/or produced by edge $e = (v_i, v_j) \in E$ is denoted as $\text{prod}(e)$ and $\text{cons}(e)$.

Figure 12: A synchronous data flow with annotated edges [SB09].

Figure 12 shows a synchronous dataflow graph with three vertices and two edges. The numbers hovering near the vertices are the number of produces or consumed tokens. As an example, A produces two tokens for B and 1 token for C. B can only fire if it has received 3 tokens from A. Since Embedded Systems are mostly iterative and non-terminating, that means it needs at least 2 iterations to fire while C can fire every time A produces a token. With this information it is possible to create a schedule involving each actor and loop it infinitely.

Edges can be delayed - in Figure 12 this is represented by a rectangle on the edge containing the number of initial tokens - meaning that the signal C receives in the $n$th iteration of the program is actually the signal A generated in the $n - 1$th iteration. This of course implies that the very first token consumed by C was not produced by A, but was already available from the start.
through an edge’s buffer [LM87]. For the precedence graph, that would mean that C can fire even before A, as all inputs it needs are already present in the buffer. This being the case, delay tokens are regarded as initial conditions in an execution, rather than part of the execution itself, thus initial tokens are produced before the execution begins [LS10].

SDF graphs can be formalized as a topology matrix (similar to the incidence matrix of directed graphs) [LM87]. This matrix represents a graph’s structure, containing one column for each vertex and one row for each edge. The value at \((i, j)\) describes the number of tokens either produced or consumed by \(j\), with production resulting in a positive value and consumption in a negative one. If the vertex \(j\) does not produce or consume any value from edge \(i\), the value is set to zero.

The topology matrix \(\Gamma\) of the graph in Figure 12 looks like this:

\[
\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 1 & 0 & -1 \end{bmatrix}.
\]

2.3.2 Homogeneous Synchronous Dataflow Graph

In this thesis, we use a further special case of a synchronous dataflow graph, the Homogeneous Synchronous Dataflow Graph (HSDF graph). It is defined as an SDF graph where for each edge \(e = (v_1, v_2) \in E\), \(\text{prod}(e) = \text{cons}(e) = 1\). This means that each vertex only produces and consumes exactly one token. A Vertex is able to fire once it has one or more tokens on all its input edges, producing one token for all its output edges in the process. [LM87, SB09].

Using the topology matrix from Section 2.3.1, it is possible to convert any SDF graph to its equivalent HSDF graph by connecting the ports so that the number of tokens each vertex sends or receives are the same as in the original SDF graph. Due to this, the resulting graph be a lot bigger than the original [SB09]. Furthermore, as an optimal periodic overlapped schedule can be computed in polynomial time from an HSDF graph [PM91], this leads to the fact that every SDF graph can be scheduled. Various works (such as
[SGB06]) explore how such a schedule can be found.

Just as the SDF, an HSDF graph is a directed multigraph $G = (V, E)$ with edges $e = (v_i, v_j) \in E$, $v_i, v_j \in V$ with a number of initial tokens (delay) $d(e)$ or $d(v_i, v_j)$. A sequence of edges $(e_1, e_2, \ldots, e_n)$ with $e_i \in E$ and $\text{snk}(e_i) = \text{src}(e_2)$, $\text{snk}(e_2) = \text{src}(e_3)$, $\ldots$, $\text{snk}(e_{n-1}) = \text{src}(e_n)$ is called a path $p$. A path includes edges as well as subpaths (i.e. the path from $\text{src}(e_1)$ to $\text{src}(e_{n-x})$ where $x > 0$), originates at the vertex $\text{src}(e_1)$ and terminates at $\text{snk}(e_n)$. If $\text{src}(e) \neq \text{snk}(e_n)$ applies to each edge $e \in E$, then the path $p = (e_1, \ldots, e_n)$ is a \textit{dead-end path}. If the path is directed from a vertex to itself (or a vertex is included twice), the path is called a \textit{cycle}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{homogeneous_synchronous_dataflow_graph.png}
\caption{A homogeneous synchronous dataflow graph.}
\end{figure}

A homogeneous synchronous dataflow graph in the format we use in this work can be seen in Figure 13. It contains 5 vertices and 5 edges. Edges are not annotated, as each edge only produces and consumes one token. Delays are denoted by a rectangle containing the number of delays followed by the letter D. In the middle of the graph a cycle can be seen, highlighted by a blue ellipse.
2.4 Allocation and Scheduling

With homogeneous synchronous dataflow graphs as our model of computation it is possible to create a schedule by allocating vertices as tasks on the available resources. This underlying concept of the analyses in this work is explained in the following section.

2.4.1 Fundamentals of Scheduling

The scheduling of a process is one of the main problems of parallel computing [Lee89], deciding where and when a task is to be executed, allowing for multiple resources and parallel execution. It can be split in multiple parts:

- Allocating tasks to resources
- Allocating channels to resources
- Timing the tasks’ execution
- Timing when the tasks communicate

One goal of the scheduling process is to minimize the iteration period (makespan) of a program, as it is directly tied to the responsiveness of a system [EZL89], which is an important aspect of an embedded system. As the vertices of the graph are functions that operate on data provided through the edges, the model of the systems that scheduled can be classified as data-driven. The scheduler decides when a vertex should fire and on which processor the function should be executed.

Four classes of scheduling. According to Lee and Ha [Lee89], four classes of scheduling can be defined. Depending on the class of scheduling used, scheduling decisions can either be made at compile time or at runtime.

Table 1 shows these classes and whether each operation (assignment, ordering and timing of tasks) can be performed at runtime or compile time. As the firing time of actors can be determined through the HSDF graph,
Table 1: Four classes of scheduling [Lee89].

<table>
<thead>
<tr>
<th>Method</th>
<th>Assignment</th>
<th>Ordering</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>fully dynamic</td>
<td>runtime</td>
<td>runtime</td>
<td>runtime</td>
</tr>
<tr>
<td>static-assignment</td>
<td>compile time</td>
<td>runtime</td>
<td>runtime</td>
</tr>
<tr>
<td>self-timed</td>
<td>compile time</td>
<td>compile time</td>
<td>runtime</td>
</tr>
<tr>
<td>fully static</td>
<td>compile time</td>
<td>compile time</td>
<td>compile time</td>
</tr>
</tbody>
</table>

scheduling is performed at compile time, therefore we use a fully static approach in this work. Furthermore, we assume a non-terminating character, this schedule is then looped for an indefinite time.

2.4.2 Fully Static Scheduling

A multiprocessor schedule is referred to as fully static if every iteration of a task is running on the same processor. In order to calculate the allocation and timing of tasks and channels, the scheduler reads the HSDF graph and builds an acyclic precedence graph (removing delays and cycles while preserving precedence relations) and then allocates the tasks to a processor. In a fully static schedule, this allocation (as well as the timing) stays the same for each iteration.

Figure 14: Example of a schedule (of the HSDFG in Figure 13).

Figure 14 shows Figure 13 scheduled on two processors. A schedule is displayed in two dimensions: the x-axis refers to the moment in time the
task is scheduled to be executed in (time axis) while the y-axis represents the various resources (resource axis). The resources in this example are the two processors as well as an interlink bus connecting the processors. Each task is allocated to one of the processors while channels (edges between tasks that have been allocated on different resources) are allocated to the interlink bus. The schedule is fully static if in all following iterations, In1 stays on resource P2 while all the others stay on P1. If the next iteration starts at or after time step 4, the schedule is classified as non-overlapping. If it starts earlier, it is an overlapping schedule.

2.4.3 Scheduling Homogeneous Synchronous Dataflow Graphs

The goal of this work is to schedule HSDFG graphs using the mentioned static, iterative scheduling process. Through the information provided by a HSDFG, vertices can be allocated on resources as tasks. In order to time their execution each vertex $v$ needs to be associated an execution time $t(v) \in \mathbb{Z}^+$ (nonnegative integers) according to a cost model. Should the real execution time of a task be unknown, $t(v)$ can also be set to an estimate amount, leading to a compile-time estimate [SB09, Lee89]. For the purpose of this thesis, the cost model has been defined in such a way that each atomic task takes one time step while a task containing subtasks (translated subsystems) take time steps according to the amount of their children. Communication between processors is represented by assigning such a cost value to each bus-access. In this thesis, communication times tested range between 1 and 1000 time steps.

Because of the non-terminating character of HSDF programs, they are run infinitely. Firing a vertex is called an invocation with an iteration being the invocation of each vertex once. For the analysis part of this work, it is our goal to minimize the iteration period, which is the time a schedule needs to finish one iteration of the HSDF.
2.5 Metrics for analyzing parallel Systems

In order to assert how well a Simulink Model can be run on multicore architectures, various metrics need to be defined first. Using the right metric makes it possible to determine how one would benefit from parallelizing the model as well as how various actions like changing the model changed the performance.

In this work only widely used performance metrics for parallel systems are used, since a system’s performance is the main motivation behind switching to a multi-core architecture. These include the iteration period - also called makespan - mentioned in the previous section, the speed-up and the efficiency [GKKG03].

2.5.1 Makespan

The makespan is defined as the total time it takes for a schedule to finish. When used for parallel systems, it can be divided into two metrics: sequential makespan and parallelized makespan.

The sequential makespan is the execution time a task needs to finish on a single processing core. Used together with the parallelized makespan, which is the execution time on multiple cores, various other metrics like the speed-up can be calculated. The lower bound for the parallelized makespan of a schedule is the critical path of the dataflow graph (the longest delay-free path in the graph), while the upper bound is the sequential makespan [SB09].

In an embedded system, minimizing the makespan also minimized the End-to-end latency since a shorter execution time leads to an earlier output and thus to a faster responsiveness.

In this work, sequential and parallelized makespan are defined as $T_1$ and $T_i$ respectively.
2.5.2 Speed-Up

Speed-up is defined as “as the ratio of the elapsed time when executing a program on a single processor (the sequential makespan) to the execution time when n processors are available ([the parallelized makespan])” [EZL89]. Throughout this work, it is defined as,

\[ S(n) = \frac{T_1}{T_n}. \]

With this metric it is possible to assert how the execution time of a system as improved by changing the number of processing elements. In theory, the speed-up can never exceed the number of processors \( p \), the best-case. A speed-up of \( p \) would mean that the processing time of each task is theoretically divided by \( p \), in other words, \( T_p \) would be \( T_1/p \). Reaching a value greater than \( p \), this would mean that a processing unit takes less time for a task than a single processor would.

While it should not happen theoretically, it it sometimes observed. This superlinear speed-up can happen in cases where the task that is performed is better fit to be distributed among many cores, e.g. when the data is too large to fit into the cache of one processor [GKKG03].

For this work we only regard speed-up as a value from 0 to \( p \).

2.5.3 Efficiency

The last metric used in this work is efficiency. It is defined as “the average utilization of the n allocated processors” [EZL89]. The speed-up and efficiency of a single processor system are always given as 1. The efficiency value tells us how much average speedup is gained by investing in \( n \) processors, enabling us to declare how worthwhile a multicore system is for a given system.

The efficiency of running a program on a \( n \) processor architecture is given as,
\[ E(n) = \frac{S(n)}{n}. \]

Since efficiency is a percentile value, it can only reach values between 0 and 1. If the value stays at 1 as more processors are added, the system has a linear speed-up, which would mean that we have the best-case scenario described in Section 2.5.2 where \( S(p) = p \), meaning that 100% of the processor’s computation power is used for the algorithm. In practice, linear speed-up is not achievable since adding more processors increases the communication time between processors as well as the time waiting for shared resources to unlock.

2.6 Related Work

There are not many related works when it comes to analyzing concurrent Simulink models in so many views, but for the actual translation of discrete Simulink, one work in particular was of greater interest while researching for this thesis. In 2005 Tripakis, Sofronis, Caspi and Verina published a paper called Translating Discrete-Time Simulink to Lustre [TSCC05]. In this paper, they presented a method to translate discrete-time Simulink models into the synchronous language Lustre [CPHP87]. They take out discrete blocks and translate those into Lustre. Not only structurally but also the functionality of each block. This way they can preserve the behavior of the original Simulink model - meaning that, when they produce output with a Lustre program, the result is the same as in the original Simulink analysis. While - just like us - they can not state a forman preservation of Simulink semantics, as they aren’t formalized anywhere, they run a number of tests to determine if they overlap for a set of input sequences.

While we assume that each part of a Simulink model can also be translated into a discrete counterpart, their assumption is that for the modeling of a embedded controller, only the discrete-time parts of Simulink should be used. Their goal is also different from ours in that we aim to analyze
a model for feasibility of its concurrency while they aim to automate the
code-generation of embedded systems through Lustre.

This work is interesting for our case mainly because they implemented
some of the block’s functionality, giving thought to the translation of Simulink
types such as int8, single, inf, etc as well as to sample times (implemented
through a clock calculus), triggers for subsystems and different time-characteristics
of the delay blocks.

While most of this has not been possible in this work, learning from some
of their solutions could benefit the translation part of this procedure.
3 Implementation of the Simulink Translator

In order to evaluate how well a Simulink Model could be parallelized and to make assumptions about is multi-core target architecture, it has to be translated into a format which we can structurally analyze with various tools. For this, the model is parsed into a Directed Sparse Multigraph. Part of the goal of this work is to document the development and functionality of this translator.

The following section describes the way the translator works. First we take a look at the general concepts the translation works with, followed by a quick look at the general algorithm. Lastly we define how to translate specific Simulink blocks.

3.1 General Concepts

3.1.1 Assumptions for Concurrent Systems

In order to translate Simulink models and to treat them as concurrent systems, several assumptions need to be made.

Discrete Signals. There are two types of signals used: continuous-time signals and discrete time signals. A continuous-time signal is an uncountable, infinite set of numbers and values. Between the start time $t_0$ and the end time $t_\Omega$, infinite possible values of $t$ can be found. Discrete-time signals on the other hand have a finite and countable set of numbers and values. There are $n$ values for $t$ between the start time $t_0$ and the end time $t_n$. When the timestep between $t_i$ and $t_{i+1}$ approaches 0, the signal has some characteristics of a continuous-time signal [LS10].

Simulink supplies both continuous as well as discrete signals, depending on the block utilized. Some blocks, such as the Zero-Order Hold can even be used to convert a continuous signal into a discrete one and the other way around. For the sake of this thesis, every block - even continuous ones - is regarded to input, process and output a discrete signal. While we could have only translated discrete blocks, this would have constricted us severely in our
goal to analyze an industrial Simulink model.

**Iterative, synchronous model.** The model we use for this translation and its following analysis is an iterative, synchronous model, based on a homogeneous synchronous dataflow (as specified in Section 2.3.2 and Section 2.4.3). HSDF programs are run infinitely and can be scheduled using a fully-static approach.

**Fully-static scheduling.** The analyses of this thesis are executed by creating a fully-static schedule for the HSDF graph as described in Sections 2.4.2 and 2.4.3. The resources, tasks and channels can be allocated to are always $n$ processors and one interlink bus, which can process multiple inter-process communication attempts at once, which means they overlap. In order to be able to schedule the HSDF, we assume a cost model. In this work, each atomic block is assumed to take 1 time unit for its computation. A subsystem containing $n$ blocks needs $n$ time units to complete.

**Assumptions for Simulink blocks.** Various assumptions have been made for specific Simulink blocks in order to be able to translate them into a HSDF vertex. These include subsystems, busses, muxers and if-else-constructs. A list of the most important assumptions can be found in Section 3.3

### 3.1.2 Translation Depth

One of the goals of this translation algorithm was to enable the user to translate the Simulink model down to a desired architectural depth, so the correlation between architectural granularity and other variables can be analyzed.

To provide this function, an algorithm was formed that replaced all parent vertices in the current graph with its children and draw the appropriate edges. This way it is possible to go deeper inside an architecture on the fly.

Low-depth output results in less vertices with a higher complexity, since most vertices contain more than one operations while going deeper into the
model’s layers produces more and more vertices to be seen as atomic operations.

3.2 Process of Translating Simulink Models

The following section describes how the actual algorithm translating a Simulink MDL-file to a DirectedSparseMultigraph works. It aims to explain all steps that are necessary after the MDL-file was parsed into a SimulinkModel Object by using the Simulink parser contained in the ConQAT Simulink Library (see Section 2.2.2). These steps are the following:

- Creating Vertices
- Creating Edges
- Adding the first layer to the graph object
- Create delayed edges
- Unfold the graph until the desired depth has been reaches.

Each subsection will attempt to explain one bullet point. It should be noted that the steps are not completely separated from one another, as the next phase can already start within the preceding one.

3.2.1 Blocks to Vertices

The Vertex class. To be able to translate SimulinkBlock objects into vertices, the Vertex class is extended with additional functionality. Apart from standard variables such as “type”, “name” and “expression”, new variables had to be added. A list of added variables can be found in Table 2. With these adjustments, the structure of SimulinkModels can be translated into a data-flow graph.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>complexity</td>
<td>Sets the operation complexity</td>
</tr>
<tr>
<td>removeWhenUnfolding</td>
<td>When this is set, the vertex will be deleted as soon as the next layer is translated</td>
</tr>
<tr>
<td>parent</td>
<td>Points to the parent of a vertex</td>
</tr>
<tr>
<td>children</td>
<td>Set&lt;Vertex&gt; of child-vertices</td>
</tr>
<tr>
<td>subEdges</td>
<td>Set&lt;Edge&gt; of edges between child-vertices</td>
</tr>
<tr>
<td>delayBlock</td>
<td>Integer showing the SampleTime the block delayed the signal</td>
</tr>
</tbody>
</table>

Table 2: New variables in the Vertex class

The BlockToVertex algorithm. The procedure to translate SimulinkBlock objects contained in the SimulinkModel object into vertices can actually be split into two phases, requiring two iterations over all blocks contained in the model. In the first one Vertex objects are created and put into a `HashMap<SimulinkBlock, Vertex>`, which allows us to find the corresponding vertex to a block when needed (i.e. when creating edges). In the second phase, the special variables from Table 2 are set.

![Figure 15: Workflow for Phase 1 of Vertex generation.](image)

First phase of the vertex translation. In the initial iteration over the SimulinkModel, generic vertices are created for each vertex that is encountered. Only the “type” and “name” variables are set. The name is taken from the block’s name parameter while the type is set either to INPUT, OUTPUT or FUNCTION.

Defining the vertex type. A vertex in can be of one of three types: INPUT, OUTPUT and FUNCTION.
INPUT is used for vertices that correspond to environment inputs. In Simulink, these are Inport blocks, as they are used as links from external sources into the current system. It can also be applied to “From Workspace” and “From File” blocks, as they link the system to a workspace or a MAT-File respectively. OUTPUT vertices receive signals that leave the system. Analogous to INPUT vertices, the corresponding blocks in Simulink are “Outport”, “To Workspace” and “To File”.

Every block that is neither an INPUT nor an OUTPUT vertex gets assigned the FUNCTION type. These vertices use a defined expression to calculate a function. This is where actual functionality can be implemented, which was not part of this work.

Figure 16: Workflow for Phase 2 of Vertex generation.

The second phase of the vertex translation. Phase two starts once the map has been filled with each vertex. Iterating over the model again, the adjusted variables are set for each block’s corresponding vertex. For a vertex’ “complexity”, an algorithm counts the number of blocks within a subsystem. If it is empty, the complexity is set to 1. The “parent” variable is set to the subsystem containing the translated block. Each block within a subsystem is added to the corresponding vertex’ “children” set.

If the block contains more blocks (thus being a subsystem), we have to set the “complexity” and “children” parameters. For the complexity of a vertex an algorithm counts the number of blocks within a subsystem. If it is empty, the complexity is set to 0 since opening the system would leave no blocks. To find the children of a vertex, we look through vertices map built in phase one to find the corresponding vertex to each block contained within
the subsystem. Each of these vertices is then added to the children set.

The “removeUnfolding” parameter is set for each inport- or outport block that is part of a subsystem, as these will have to be removed once the graph is unfolded. If the block was a \textit{Delay} block, the sample time is extracted and set to the “delayBlock” parameter.

Finally, all lines within the subsystem are translated into edges and added to the “subEdges” set.

\subsection*{3.2.2 Lines to Edges}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{lines_to_edges.png}
\caption{Workflow followed while translating lines to edges.}
\end{figure}

The translation of lines to edges is actually overlapping with the previous step. When the “subEdge” parameter is set, the edges get created. Their “name” parameter is set to the equivalent of “SrcBlock ->DstBlock”. Just as with vertices, the Edge class was extended with one extra parameter called “vertices” which points to a Pair object containing the source- and destination-vertex of the edge. This makes adding edges while unfolding easier.

Since the first layer does not have a parent, the edges on that layer do not get translated in this step. This actually happens in the next step, while creating the graph.

\subsection*{3.2.3 Creating the Graph}

The shortest step is the actual creation of the graph. After the creation of a \texttt{DirectedSparseMultigraph} object containing vertices and edges, all that
Create DirectedSparseMultigraph object
Add vertices from the first layer of the model
Add outgoing edges of each vertex

Figure 18: Workflow while creating the graph object.

is left is to add these.

Since we have a map of all vertices, we could add those to the graph. This way we would not be able to define the translation depth. Instead we only add all vertices from the first layer to the graph, while also translating each block’s outgoing line into an edge and adding it to the graph.

3.2.4 Adding Delays

Find vertices in graph object where "delayBlock" > 0
Delete vertex from graph
Connect preceding and following vertices with new edge
Assign "delayBlock" value as delay

Figure 19: Workflow when converting delay blocks to delayed edges.

Delays are variables that define how many time steps the signal that is transported over the corresponding edge is held. Since delays are implemented as blocks in Simulink, we can not translate delays directly.

Instead we translate each delay Simulink block as a vertex that has it’s delayBlock variable set to the sample time of the block. Every vertex that has its delay parameter set is then delete and its incoming and outgoing edges are merged together into a new Edge by connecting the incoming edge’s source vertex with the outgoing edge’s destination vertex. We then add the delay value to that edge.

With this the first layer of the Simulink model has been translated.
3.2.5 Unfolding the Graph

Figure 20: Workflow when unfolding a graph.

If the first layer of the model was not the desired output, it is required to “unfold” the graph. This translated the next layer of the model, thus advancing to a deeper architectural granularity.

The Unfolding Algorithm. Unfolding the graph does not need the SimulinkModel object anymore, so this step can theoretically be done way after the model has been translated. At this time it can not be reversed, though, so it should be used with care.

For each of the graph’s vertex containing children, these children are added to the graph as vertices. The same is done with all the edges in the subEdge set. With this step finished, the parent vertices can be deleted, since they have now been replaced with their children.

Since this translated inport blocks that were contained in a subsystem as vertices, we will have to delete these. For this we use the same algorithm as for creating delays (see Section 3.2.4). After finding all vertices that have the removeWhenUnfolding parameter set, these vertices are deleted and all their incoming and outgoing edges are merged.

This procedure is repeated until the desired translation depth has been reached.
3.3 Translating Specific Block-Classes

In the scope of this thesis, only the structure of a Simulink model is translated. Functionality has not been considered in most cases. The Simulink standard encompasses a great number of different blocks which can be further configured and combined. Thus only the most-used blocks - as identified by analyzing a Simulink model used in a real-life Embedded Software System (see Table 3) - have been translated in detail while some are only present in a structural sense.

<table>
<thead>
<tr>
<th>Block</th>
<th>No. of Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>InPort</td>
<td>339</td>
</tr>
<tr>
<td>OutPort</td>
<td>195</td>
</tr>
<tr>
<td>SubSystem</td>
<td>167</td>
</tr>
<tr>
<td>Constant</td>
<td>163</td>
</tr>
<tr>
<td>DataTypeConversion</td>
<td>76</td>
</tr>
<tr>
<td>Logic</td>
<td>66</td>
</tr>
<tr>
<td>Switch</td>
<td>65</td>
</tr>
<tr>
<td>Relational Operator</td>
<td>61</td>
</tr>
<tr>
<td>BusSelector</td>
<td>58</td>
</tr>
<tr>
<td>Reference</td>
<td>44</td>
</tr>
<tr>
<td>Selector</td>
<td>41</td>
</tr>
<tr>
<td>Sum</td>
<td>38</td>
</tr>
<tr>
<td>ActionPort</td>
<td>30</td>
</tr>
<tr>
<td>Product</td>
<td>25</td>
</tr>
<tr>
<td>BusCreator</td>
<td>23</td>
</tr>
<tr>
<td>UnitDelay</td>
<td>22</td>
</tr>
<tr>
<td>From</td>
<td>19</td>
</tr>
<tr>
<td>Gain</td>
<td>19</td>
</tr>
<tr>
<td>Demux</td>
<td>14</td>
</tr>
<tr>
<td>MinMax</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3: Twenty most-used Simulink Blocks in a real-life industry example

There are some exceptions to this rule, as some blocks such as, but not limited to, Memory (Rank 21), TriggerPort (Rank 26) and EnablePort (Rank 34) which had to be translated in greater detail as they are either important
for structural integrity or needed for our cause.

3.3.1 Sources and Sinks

The InPort and OutPort block. The most used blocks in Simulink models are InPort and OutPort blocks. They receive and provide signals from outside systems. When placed inside a SubSystem, they are linked to one port of the SubSystem block. The InPort Block then forwards the signal received to this port to each block it is connected to while the OutPort block forwards the signals it receives to the output port of the Subsystem.

Functionality in translation. Since they are used as a link to outside systems, they are translated into INPUT and OUTPUT vertices respectively. In the finished, drawn DFG, they are distinct by their dotted outline.

Port-Blocks inside SubSystems can not be translated this way as INPUT vertices can not have any incoming edges while OUTPUT vertices can not contain outgoing ones. When unfolding the graph, so the next layer of blocks is translated, these blocks are removed from the graph, as the only way to keep them would be to translate these internal InPorts and OutPorts as FUNCTION vertices.
Considering that all In- and OutPort Blocks do is forwarding signals to other blocks without actually calculating anything, this would lead to random noise in the analyzation, as they would have a cost-value assigned. They would also slow the analyzation process down considerably, as translating them artificially increases the amount of tasks.

For the translation process, they are translated as temporal FUNCTION vertices while unfolding a graph. In the last phase of the translation each vertex that originates from an internal Port block is deleted from the graph and its preceding and following vertex are connected by an edge - thus keeping the structure of the model intact.

![Constant block diagram](a) A model containing a Constant block (b) Figure a) translated into a DFG block

Figure 22: Example of a Constant Block’s Translation

**Constant block.** Another often-used source block is the *Constant* block. It generated a specified value and forwards it to all adjacent function blocks. While it is classified as a source block in Simulink, it is not translated in the same way an Input block is. Since it does not provide input from any outside system but generates specific input itself, it is not considered to be the equivalent of an INPUT vertex but rather a FUNCTION vertex. As such it is assigned a complexity value according to the cost-model used.

### 3.3.2 Function Blocks

As stated in Section 2.1.1, function blocks are blocks that take one or more signals from their inports, apply a function and output the result through
its outports. Function blocks are one of the most-used block classes. Most blocks of this class are translated as FUNCTION vertices with ingoing and outgoing edges. Since functionality, as well as different complexity computations for different operations, have not been implemented as of the writing of this document, each function block is essentially translated the same way. In this section we are going to take a look at the following often used function blocks, explain their functionality and their translation into a DFG.

(a) A model containing a Data Type Conversion block  
(b) Figure a) translated into a DFG

Figure 23: A model using a Data Type Conversion Block and its translation

**The Data Type Conversion block.** In order to convert input from one data type into another, Simulink provides the *DataTypeConversion*-Block. This block translates its input into the data type specified in the *Output Data Type*-Parameter. It also handles different rounding algorithms for integers such as floor, zero, nearest and so on. As most other function blocks it is translated as a FUNCTION vertex.

**The Logic block.** The LogicalOperator-block, seen in Figure 24 can perform a specified logical operation on the input. When comparing input values, it handles them as TRUE if they are nonzero and FALSE if they are zero (for boolean types it just takes the value). The type of operation can be specified in the block properties using a dropdown menu. It supports AND, OR, NAND, NOR, XOR, NXOR and NOT. All operations except for NOT take multiple inputs and produce a single output.
Translating the block works as any other function block. It is translated as a structural FUNCTION vertex with incoming as well as outgoing edges.

The **RelationalOperator block**. Working similar to the LogicalOperator block detailed just now, the RelationalOperator compares two inputs using a specified relational operator. It works by taking the signal received on the upper port, assigning the operator and matching it with the signal from the lower port. The operators used include equal (\(==\)), not equal (\(\neq\)), less than (\(\lt\)), less or equal (\(\leq\)), greater (\(>\)) and greater or equal (\(\geq\)). It also supports a one-input mode where it provides the operators isInf, isNaN and isFinite. It is translated similar to the LogicalOperator.
The Sum block. The Sum block works like the Add block in that it adds or subtracts two or more signals and returns the result as its output. There is only a small difference between the two blocks, which results mainly in the fact that the Sum block seems to be replacing the Add block, as no documentation of the latter is to be found in the online Simulink reference at [Mat11b] at the time of this writing. Both blocks support addition, subtraction as well as terms such as $z = x_1 + x_2 - x_3 + x_4$. This can be specified by setting the number of inputs as well as the operation for each port. The example we just mentioned would be specified by setting $++-+$ as the “List of signs” parameter. As most function blocks it is translated as a FUNCTION vertex as seen in Figure 26.

The Product block. The Product block implements two other basic arithmetic operations: multiplication and division. It works and is translated similar to the Sum block, in that it takes two or more inputs and is able to string operations together. As a default, the block uses two inputs and is able to mix and match scalar and non-scalar elements (i.e. $2 \times [4 \ 8] = [8 \ 16]$). By using the “Number of inputs” parameter, strings such as $*/*$ can be set enabling the block to implement a function such as:
The Gain block. The Gain block multiplies the input - which can be either a scalar, vector, or a matrix - by a constant gain value. It supports both element-wise gain as well as matrix gain. When translated, it is represented by a FUNCTION vertex with one input and one output.

The MinMax block. In order to output the minimum or maximum value of one or multiple inputs, Simulink provides the MinMax block. If the block only has one input port, the input must be a vector or scalar. For multiple inputs, the dimensions for each must be identical. The block then outputs a
A model containing a MinMax block

Figure 29: Translation of a MinMax block.

scalar equal to the minimum or maximum element of the input vector, depending on which mode has been specified. It is translated as a FUNCTION vertex with a variable number of incoming edges and one outgoing edge.

3.3.3 Delays / Discrete Data Storage

Delay blocks. Adding a delay block between two blocks delays the input of the following block by a specified sample time (iteration period). This makes them invaluable for translating self-loops, as Simulink does not allow a block to point to its own input port without passing another block in the process.
While UnitDelay and Zero-Order Hold are used differently in Simulink, they are treated the same way in this work, as the only difference between them is the type of signal they can receive and output. Since we assume that only discrete signals are used and each block can work with them, a distinction between them is not needed.

**Translation process of delay blocks.** Since the DFG implementation used in our translation accepts delayed edges, these blocks are no longer needed in the translated graph. In the DFG, an edge can be delayed, thus displaying a square on the edge containing the number of iteration periods it is held before used.

In our translation, delays are translated in a way similar to internal InPorts and OutPorts. They are translated as FUNCTION vertices with a delay value first. In the last step they are deleted together with the internal ports and edges are drawn between their neighbors. The delay value is then added to these edges as seen in Figure 30.

For a Memory block, this value is always 1, while for UnitDelay and Zero-Order Hold blocks this value is specified by the sample time in Simulink.

### 3.3.4 Structural Blocks

As Simulink models can get rather big, structural blocks are provided to make it easier to read the model with human eyes. They are also used to structure a model according to the target architecture it will run on. The most-used of these blocks is the SubSystem block.

**Translating the SubSystem block.** Since the SubBlock has no real function besides containing other blocks, thus acting like a folder, the block does not have to be translated into a vertex. Instead its borders are opened and the containing blocks are directly linked to the Edges pointing to or away from the SubSystem.
As seen in Figure 31, the In- and Outports within the SubSystem have been removed, since they were only needed to connect the ports of the Subsystem block with the contents of the block. Instead, the edges point directly to the vertex that was translated from the block following the InPort block. The same was done exiting the subsystem.

Special Subsystems As explained in [label:todescription] Enabled SubSystem is a sort of special subsystem that contains an EnablePort block as a port. Since it is not possible to discern the value of the signal the block receives on its EnablePort, this block can not be translated in the same way it functions in Simulink.

It is instead translated in the same way as a normal subsystem. When the translation depth is on the layer containing the enabled subsystem, it is translated as a standard vertex with one extra incoming edge. Unfolding further reveals the contents just as a normal subsystem, with the difference that the EnablePort is not erased in the same way inports and outports are.

The reason for this is that the argumentation for deleting inports and outports does not apply on the enable port. While the former are just representations of Simulink ports in subsystems, the latter contains a function describing what has to happen when it receives a specific signal, thus making it a FUNCTION vertex.
The same applies for the Triggered SubSystem and the Action SubSystem. While the trigger has a specific function, we can at the current time not discern what it does, only that it does have to do something. Thus the TriggerPort and ActionPort can not be deleted.

Library References. Another special kind of subsystem, that is also translated in the same way is the Reference block. It references a Simulink library, which contains more blocks. What sets libraries apart from normal subsystems is their reusability. A library can be inserted at multiple places whenever needed and is independent from its environment apart from the inputs and outputs that need to be of the right type. As references are actually Simulink blocks pointing to another MDL-file, they are translated the same way as a SubSystem block.

3.3.5 Signal Routing Blocks

Signal Routing blocks are provided by Simulink to merge, split and route signals from one block to another. The most-used blocks of this types are busses, muxes, switches as well as the From and Goto block pair.

Busses and muxers in Simulink. Nusses in Simulink are used to merge signals together into a vector (BusCreator) or to take specific signals out of a formerly created bus (BusSelector).

While the Mux and Demux blocks provide similar functionality, they differ in the aspect of signal management. See Section 2.1.1. Since the decision which signals to extract is made in the modeling process and is already specified before the translation, these two blocks work exactly the same in regard to their translation.

Translating busses and muxers to vertices. Translating busses essentially works like any other block. It is converted into a FUNCTION vertex with all its incoming and outgoing signals being translated into edges.
Another theoretic way to translate this block would have involved the same mechanism used for InPort and OutPort blocks within subsystems: only translating the signals but skipping the block. The decision to translate bus blocks as vertices was made because while busses only route signals to other blocks just like out- and inports, they also provide functionality: they either add signals to a vector or extract specified signals from it. This operation costs resources and thus time. So the decision to translate them has been made.

The same logic also applies to muxers and demuxers. Adding to and reading from a vector is a costly operation which is why they have to be individually translated.

The Switch block. The Switch block takes three inputs. Only the first or the third input are able to pass through, depending on how the second input compares to the specified logical expression. Because of this the first and third inputs are called data inputs and the second one is called the control input. For the control input, three possible settings exist: \( in2 \geq \text{Threshold} \), which checks whether the control input is greater than or equal to the specified threshold value, \( in2 > \text{Threshold} \), checking is the input is
greater than the value and $in2 = 0$, which routes the signal through the first data input if the control input is not zero.

When translated, the block is represented by a FUNCTION vertex with three incoming edges and one outgoing edge. While it could have been deleted and only translated as a edge, we decided against this approach since the logical operation needs to be accounted for.

![Diagram](image.png)

(a) A model containing a From and a Goto block  
(b) Figure a) translated into a DFG

Figure 33: Translation of From and Goto blocks.

**The From and Goto blocks.** From and Goto are two connected blocks that send and accept signals. The Goto block can send the signal it receives to various From blocks throughout the model while each From block can only accept a signal from exactly one Goto block. In its presentation, each From and Goto block has a tag name enclosed in brackets. The relationship between both blocks can be read from these tags, as there is always only one Goto block per tag name and one or multiple From blocks. This enables communication through different architecture layers.

This comes with a problem concerning our translation. While From and Goto blocks could be translated by drawing an edge from the Goto block to each From block it is connected to, this would not be possible for blocks on different layers, as our unfolding algorithm actually does not know the
vertices contained in deeper layers. In the time given it was also not possible to implement a way to check after each unfolding if a new From and Goto pair can be connected. Another problem is, that the lookup operation a Goto block has to compute to find each From block also has a complexity, so the blocks had to be translated as vertices. That is why we translated these blocks as FUNCTION vertices right now, defining the communication between the two blocks as “functionality of the block”, which is not part of this work.
4 ConQAT Integration

4.1 Why to use ConQAT?

As explained in Section 2.2, ConQAT is used to continually analyze software quality. One of its advantages is, that it works automatically without user interaction [DS06]. This means that we are able to run the analyses we need at any time in the development process. Since the analyses (the allocation and scheduling algorithms) covered by this thesis can take several hours, days or more, depending on the number of tasks that need to be allocated, this is the main advantage ConQAT has for our cause.

With an offline tool like ConQAT, developers can run these parallelism analyses periodically in a nightly or weekly build, using the results to make decisions in their development process.

In this thesis, we picked ConQAT also because of the sometimes long computation process but because of ConQAT’s expandability as well. Since ConQAT analyses are build out of processors, that can be arranged in any way we need, analyses can be customized and new functionality such as quality assessment can easily be implemented later on.

This section explains the processors that have been developed for this analysis. Please note that due to time constraints, the goal was only to have the general procedure implemented.

4.2 The SimulinkDataflowTranslation Processor

The first processor we wrote for our analyses is the SimulinkDataflowTranslation Processor seen in Figure 34.

This processor takes a SimulinkModel-object as its input and translates it into a Graph-object. For this it uses the algorithm explained in the previous section. The input is provided by a SimulinkScope processor already provided by ConQAT, which takes a specified folder and translates every MDL-File into a SimulinkModel-object (ISimulinkElement) using the parser library de-
tailed in Section 2.2.2 and then outputs it as a tree of SimulinkModel nodes. Since we only aimed to analyze one MDL-file in the course of our case-example, we implemented our own SimulinkScope processor for testing, which - instead of being provided with a folder - is provided with a MDL-file directly and translates it into a single SimulinkModel/ISimulinkElement object, which it outputs. Since it actually only changes the input and output file type, this processor is not detailed here.

4.3 The DataFlowScheduleProcessor

The SimulinkDataflowTranslation processor is then connected to the SimulinkDataflowScheduler processor. This processor takes a graph and allocates and schedules each vertex as a task, using the concepts detailed in Section 2.4. The left input connector receives a Graph-object from the SimulinkDataflowTranslation processor while the second input connector offer access to the three variables Granularity, Number of Processors and Message Speed. As defaults they are set to 3, 2 and 100 respectively. The processor then computes a minimized schedule for the given graph and architectural constraints and outputs the minimized makespan.
5 An Industrial Case-Example - The ACC System

5.1 Motivation

The Goal/Question/Metric approach. The QCM approach to software metrics defines a six-step process for identifying the right metric to reach a specified (software) goal [Bas92]. First a goal has to be developed. For this thesis, we decided on responsiveness of the system as our goal, since one of the reasons multicores are used is the performance gain and reactivity boost we receive by utilizing them. We define responsiveness as the time between user or component inputs - which is actually the iteration period of the system.

Better responsiveness. This means that in order to maximize a system’s responsiveness with a given architecture, we need to minimize the iteration period of the computation - its makespan. In this section, we will analyze a industrial Simulink model using the metrics defined in Section 2.5 to identify the level of concurrency in the model and show limitations.

As stated in Section 1.1 (Figure 1), we will analyze the model in four dimensions. The number of processors, communication costs between components/cores, architectural granularity/translation depth and from an economical standpoint. While the last point is not part of the analyzation process per se, it will be considered in the final evaluation of the results.

Four dimensions of the analysis. As stated in Section 1.1 (Figure 1), we will analyze the model in four dimensions. The number of processors, communication costs between components/cores, architectural granularity/translation depth and from an economical standpoint. Increasing the number of processors is the first thing that comes to mind when aiming to minimize the iteration period through scheduling of tasks on multiple processors. It is our goal to show how well a model scales with the increasing number of cores as well as under the influence of the other factors.
Architecture Granularity / Translation Depth. One of these factors that we wanted to explore was the influence of granularity on the concurrency of a program. Since a model is normally not a sequence of atomic operations, but rather a of more complex computations chained together. In Simulink - considering our cost-model - these larger computations are represented by SubSystem blocks. Each subsystem contains various atomic blocks chained together as well as other SubSystems. Translating a model down to the last layer increases the granularity of the model to 100%, as each vertex contained in the model after unfolding has a complexity of one. While this will - as seen in the results - enable us to schedule more tasks on different processors, at some level it will not make as much of a difference anymore. Raising the number of tasks also slows down the allocation and scheduling considerably, maybe requiring a different time-step for each analysis. The actual system might even take more time to run on a real system as well, considering that communication is quite expensive.

Communication cost. This brings us to the next important factor, the cost of inter-processor communication. Whether it is through busses or shared memory, it is one of the, if not the biggest bottlenecks when dealing with performance of multi-core architectures [CGP07]. In a realistic system, communication can take a long time and use a big chunk of the iteration period. This is why a scheduler will sometimes not allocate tasks on different processors if they still need to communicate, as this would impact the makespan in a greater way. This, of course, is also dependent on how the system was modeled. A system that was made with concurrency in mind could minimize these communications between tasks and thus allow further allocation on more cores.

Economical view. As we already discussed briefly in Section 1.1, this also means that these factors influence the number of processors that are actually feasible for a system. While it would still be possible to use a system with a higher number of cores, the case that some of these cores would not be utilized at all. This is where the economical view of these analyses comes
in. It helps us to make an informed decision on whether an investment in a multi-core architecture with i.e. eight cores is a good idea or if less processors, and thus less price per unit, are enough to implement the system efficiently.

**Parallelism as an architectural driver.** For this we translate and analyze a industrial model. But it should be kept in mind that there is a motivation behind every architectural decision - a goal of some sort. It could be that the system is supposed to work well on a specific architecture or an architecture type or that the system is supposed to be easily reusable as standard architecture. One such driver would be parallelism - making a system work well on multi-core architectures.

The model used in this case-study is a pre-existing model that has probably not been developed or optimized with parallelism as its driver. This means that the results of our analysis should be viewed with this in mind.

### 5.2 The Adaptive Cruise Control Simulink Model

**Choosing the ACC-Model.** Instead of analyzing the whole Simulink model that was provided, we decided to take out a distinctive part of it and translate as well as analyze it. The reasons supporting this decision are numerous. First of all, it was essential to reach an acceptable processing time for the analysis algorithm while still utilizing an usable part of the model to receive meaningful data. Translating the whole model would have led to a high number of vertices in the upper layers, thus preventing us from reaching the deeper ones in the time provided by the constraints of this thesis. To remedy this, it has been decided to take out the part responsible for the *Adaptive Cruise Control* function as it is rather independent from the rest of the system. As seen in Figure 36, it also contains enough delayed edges, weakly connected components and cycles to be feasible for analysis. It also contains most of the blocks that have been featured in Section 3.3.

**ACC-Model description.** The model translated and analyzed in this case-study is actually only part of the greater industrial model referenced in
Figure 36: The translated ACC-System with 164 vertices and 276 edges.

Table 3, specifically one SubBlock. It contains the logic for an ACC-function. It contains 672 blocks, 70 of which are SubSystems, containing more blocks up to a total depth of 6 layers. The blocks are connected using 1624 edges with 10 being delayed by UnitDelays, which results in a rather low - but still viable - count of cycles and a lot of weakly connected components as seen in Figure 36.
**Adaptive Cruise Control.** As stated above, the part of the system we analyze handles the Adaptive/Autonomous Cruise Control (ACC) mechanism. The ACC is a special case of a cruise control system, regulating the car’s speed not only by accelerating and decelerating but also by using the breaks when needed. To do this, the ACC-System utilized a sensor (radar) to measure the distance from the vehicle in front of the driver. It can be utilized for both automatic and manual transmission - for manual vehicles it sends an acoustic and/or visual signal to the driver, signifying him to change the gear [LCJ95].

The ACC-system, when utilized, is a high priority system. A user expects it to always regulate their speed accordingly and ready the breaks when needed. Dependency on the system might lead to accidents if a computation aborts or takes too long. This is another reason for choosing the ACC-System. Because of its characteristics, the responsiveness of the system has to be as high as possible.

### 5.3 Experiment Setup

In order to run these analyses, we used different hard- and software. The computers that were used to run it were different personal computers with Intel multi-core processors ranging from 2 to 4 cores and 1.81 to 4.4 GHz, with a RAM of 2 to 8GB. The analysis was run on multiple computers at the same time, enabling us to finish the procedure faster. A single run of the algorithm took between 0.5 seconds for lower layers as high as 46 hours for some of the lower layers.

On the software side, we use a the ConQAT processors detailed in Section 4. The SimulinkScope delivers the MDL files and parses them into SimulinkModel objects that can be read by our translator. The Simulink-DataflowTranslator then translates these into Graph objects containing layer 0 (the model itself) translated into a graph. The unfolding Algorithm to reach down to the desired layer is then actually used in the SimulinkDataflowScheduler processor. It takes two inputs: the graphs as well as the parameters
(depth, processors, communication cost) we want to analyze it with. As results, the minimized makespans are then written (plainly) on a HTML page.

![Diagram](image.png)

Figure 37: The Layout of the Analysis in ConQAT.

The parameters used in this analysis were the following: For the processors, we used 1, 2, 4, 8 and 16 cores. The communication cost was set for each at 1, 10, 100 and 1000 time steps. All of these cases were inspected at granularity levels 1 to 4.

### 5.4 Experiment Results

After around two weeks of intensive testing, we were able to attain interesting results. An excerpt can be found in the following tables - the rest of the
results can be found in the appendix.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230</td>
<td>230</td>
<td>306</td>
<td>327</td>
</tr>
<tr>
<td>Depth 3</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>Depth 4</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
</tbody>
</table>

Table 4: Minimum makespan results for 2- and 4-core architectures.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230</td>
<td>230</td>
<td>306</td>
<td>327</td>
</tr>
<tr>
<td>Depth 3</td>
<td>77</td>
<td>77</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>Depth 4</td>
<td>51</td>
<td>51</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 5: Minimum makespan results for 8- and 16-core architectures.

<table>
<thead>
<tr>
<th></th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Layer 2</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>Layer 3</td>
<td>63</td>
<td>94</td>
</tr>
<tr>
<td>Layer 4</td>
<td>164</td>
<td>276</td>
</tr>
</tbody>
</table>

Table 6: Number of vertices and edges per layer.

5.4.1 General Observations

The first thing we can see is, that the jump from two to four cores is very significant if we look at layers starting at three. We have a 1.98 speedup for utilizing two cores at a translation depth of three or four. On 4-core architectures, this is dependent on the communication delay. For a minimal delay, the speedup is 3.98 while for a higher cost it amounts to 3.63. This
means that for 2-core architectures we’re looking at an efficiency of 99% while the efficiency of this model running on a 4-core architecture is between 91 and 99%. For layers 1 and 2, the makespan on all architectures is the same, which means that on these layers, the critical path is too long to ensure a high concurrency rate.

<table>
<thead>
<tr>
<th>Layer</th>
<th>1 Core</th>
<th>2 Core</th>
<th>4 Core</th>
<th>8 Core</th>
<th>16 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Layer 2</td>
<td>327</td>
<td>306</td>
<td>306</td>
<td>306</td>
<td>306</td>
</tr>
<tr>
<td>Layer 3</td>
<td>327</td>
<td>165</td>
<td>90</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>Layer 4</td>
<td>327</td>
<td>165</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

Figure 38: A Chart of the shrinking minimum makespan at 100 message cost.

Figure 38 shows the shrinking minimum makespan on each layer with the rising number of processors available. The cost for communication between processors is 100, which is the most realistic of the four. On layer 1, the makespan stays at a static 327, which is the complexity of the whole ACC-system. This is the case because on layer 1, the graph contains two subsystems that are directly adjacent to each other and can not be parallelized. This always results in the sequential makespan of the system. On layer two, we gain a few atomic blocks, as well as more subsystems that can be run concurrently. Most of the system can still only be run sequential,
as the jump from 2 vertices and 1 edge to 12 vertices and 20 edges has not produced enough concurrent parts of the graph.

The big jump happens at layer three. With now 63 vertices and 94 edges to allocate, the minimum makespan jumps from 306 to 165 on dual-core architectures (resulting in that 1.98 speedup and 99% efficiency mentioned earlier) and to 90-91 on 4-16 core architectures (resulting in speedups of around 3.6 and an efficiency of 90% for 4 cores, 45% for 8 cores and only 22.5% for 16 cores). On layer four, the makespan stays constant, even though the number of vertices and edges nearly triples to 164 vertices and 276 edges. The speedup as well as the efficiency undergoes no change from the way it was in layer 3.

While a quadcore architecture seems to be the most efficient from this data, we now take a look at an octacore architecture in detail.

![Analysis for 8 Cores](chart.jpg)

Figure 39: A Chart of the makespan with fixed 8 cores.

In Figure 39, we can recognize the results we discussed just now by looking at the “Message cost 100” column. Here, the makespan for layer 3 and 4 is exactly the same - apart from a small discrepancy that results
from the characteristic of the scheduler to drop the analysis once ($\Delta = maximum Makespan - minimum Makespan < 3$). Earlier we found that at message cost 100 the makespan of a system with granularity level 4 on a quadcore architecture is the same as on an octacore architecture. Figure 39 shows that for message costs 1 and 10, the makespan on octacore architectures lowers to 51, leading to a speedup of 6.41 and an efficiency of 80\% (as compared to the 45\% on systems with message cost 100). This shows that lowering the message cost could enable the use of an efficient octacore system. It can be seen once more that increasing the granularity to layer 4 does not have an effect on the minimum makespan of this system.

5.4.2 Evaluation of the Results

Using these results, decisions can be made on the kind of architecture to use for different situations, considering the efficiency, makespan, speedup as well as the rentability of the system.

Choosing the number of processors. According to these findings, an architecture utilizing 4 processors seems to be the most feasible for this system. Even on high latency systems, the efficiency of the 4 cores does not drop below 90\%. An 8-core architecture would only be feasible for low latency systems (message cost 1 and 10), leading to a speedup of 6.41 in contrast to the 3.9 speedup a 4-core system receives under the same conditions. This leads to an efficiency of 80\% for the 8-core system and an efficiency of 98\% for the 4-core system, though. So even at lower latencies a 4-core system might be the better choice considering every aspect, including finances. Depending on the price of each processor, those extra 2.51 points of speedup might not be worth the expense - even more so when considering the higher price for such a low latency system that actually allows this kind of speedup on a 8-core architecture in contrast to a low latency system. The price per unit (PPU) can also be minimized using concepts like economies of scale such as a lower PPU for mass-orders, if the resulting system is supposed to go into mass-production. Using a 16-core architecture with this system is never efficient, as is always produced the same makespan as the 8-core architecture,
resulting in an efficiency value of only 40% on low latency and 23% on high latency systems.

**Choosing a granularity level.** The choice of how granular the architecture should be is very dependent on the number of processors available. On 4-core systems, the choice should obviously be at least down to layer 3, considering that the speedup rises from 1.07 on layer 2 to 3.6 on layers 3 and 4, raising the efficiency from 26% to 90%. As stated earlier, on low latency systems an efficiency as high as 98% can be attained both on layer 3 and layer 4 architectures. Considering that the efficiency is the same, the choice should fall on the lower architecture, as the overhead of the scheduler increases with a rising granularity [CSY90]. Presuming that a low latency system is used and an 8-core architecture has been deemed efficient and rentable enough to be used, the choice which granularity to utilize falls upon level 4. As seen in Figure 39, the makespan falls another 26 time units for low latency 8-core systems on level 4 compared to a low latency system on level 3. This results in an increase of the speedup from 4.25 to 6.41 - an efficiency increase of 27% (53% compared to 80%).

Another reason to choose a lower granularity level is the analysis procedure itself. As seen in the execution times found in the appendix of this work, the analyses on layer 3 took between 13.6 seconds and 15.6 minutes to complete for each test while the same procedure on level 4 took between 2.4 and 46.9 hours, resulting in very high increase of the computation time. Depending on the time (and money) spent on these analyses as well as the kind of build that is aimed for - whether it is nightly, weekly, monthly or bi-monthly - a decision has to be made that includes a number of tasks that can be analyzed in the given time period. For this system, a daily build would only be possible on layer 3 and above, while layer 4 would be better analyzed in a weekly build (considering the number of analyses as well and depending on the number of parallel computed analyses). Layers below the fourth one have not been tested thoroughly, but one such analysis was run and did not deliver results even after 3-4 days.
Choosing a communication topology. As already stated, the message delay has a high influence on the makespan - and thus the concurrency in general - of the system. While it is not efficient to use a 8-core processor on systems with message cost 100 or 1000, it becomes a serious alternative to the 4-core architecture once the message cost falls as low as 10 or even 1 time steps. While a communication cost of one is rather unrealistic, a cost of 10 might be possible through optimization. Depending on the cost of these optimizations it might be worthwhile to invest in them to be able to reach a higher speedup - depending on the type of system that is being built. Other than the choice of processors and granularity, this is a choice limited by the hardware. Once the system reaches a certain size, communication overhead between components will always rise [CSY90, GKKG03, Lee89].

5.4.3 Conclusion

This section showed how, using the analyses detailed in this thesis, informed decisions about the type of the system used can be made. Considering economical factors and technical factors and evaluating a system throughout its development using those can lead to a system optimized for performance on concurrent systems as well as a financial gain by building exactly the system that is needed instead of overshooting the specifications.

As an example we inspected an industrial Simulink model and were able to come to several decisions on the factors that should be focussed on in development - keeping the message cost to a minimum using 8 cores with a granularity of level 4 to gain further speedup or, if the latency can’t be kept low, to utilize 4 cores at a granularity level of 3. These scenarios provided the highest efficiencies at 80% to 98%.
6 Discussion and Future Work

6.1 Discussion

Some parts of this thesis are still open for discussion, as they are either sub-optimal or need to be revised. Namely the cost model used for the allocation and scheduling algorithm as well some of the assumptions made to simplify the translations of Simulink models.

6.1.1 Cost Model

The cost-model we decided on in this work is rather simple. Since we have not been able to translate the functionality of Simulink blocks into functionality of vertices in synchronous dataflow programs, we assigned a static cost model. Each atomic operation needs one time step for its computation. For subsystems, we count the number of atomic blocks contained within them as well as the number of blocks contained within subsystems within subsystems recursively and assign this number as their cost. For example a block containing 3 blocks: A an InPort block, a SubSystem block containing 3 blocks and an OutPort block will have a complexity of 6. Since each atomic block takes one time step, this block will take six.

While this cost model works, it is neither efficient nor really realistic. The problem is, that each vertex, where it represents a simple arithmetic expression such as \( x = y + z \) or a more complex vector operation, takes one time step. In reality, blocks like the BusCreator, Switch, Ground and advances operators like the Discrete-Time iterator, are far more complex and take far more time to compute than a simple arithmetic operation.

There are two ways to remedy this problem: Define a static complexity for each block (i.e. \( \text{complexity}(\text{Sum}) = 1, \text{complexity}(\text{BusCreator}) = 5 \)) or do a precise complexity analysis or at least an approximation for each block. We decided that the first solution actually has the same problems our simple cost model poses. It is not realistic, as most of these values would be approximated subjectively and not efficient, as these values could not be
assigned automatically.

The second solution would be ideal but could not be implemented in the course of this thesis. For thoughts on this feature, see Section 6.2.2.

6.1.2 Assumptions for translating Simulink Models

In this work we made assumptions for the translation of various blocks. Most of these assumptions had to do with continuous blocks such as “memory”. Simulink implements both discrete and continuous signals. In order to have both be present in models at the same time, it provides functionality in their data storage/delay blocks to translate continuous into discrete signals and the other way around, which is the reason multiple delay blocks (Memory, Zero-Order Hold, UnitDelay) exist. We also made assumptions when translating busses, muxers and the From and Goto blocks.

In order for this translation to work with our implementation of a synchronous data flow, we had to either make these assumptions or decide to only translate models that only use discrete blocks. In order to be able to include a wider array of models, as well as to actually translate the model provided to us, we decided that these assumptions were necessary.

Concerning the assumptions made for the signal routing class of Simulink blocks (see Section 3.3.5), some of these had to be made out of time constraints while some where needed to be able to provide functionality. Translating the From and Goto blocks as simple FUNCTION vertices and not as edges connecting the blocks referenced was the only solution available to us considering our need to be able to unfold a graph to any layer we desire since drawing edges only works on blocks placed on the same layer.

The assumptions for BusCreators as well as BusSelectors on the other hand had to be made out of time constraints. And could be solved by implementing functionality of Simulink blocks in a data flow program. More on this is detailed in Section 6.2.1.
6.2 Future Work

6.2.1 Vertex Functionality

In this work, we only translated the structure of Simulink models and assigned a static cost to each block. This was a decision driven by time constraints, as the analyses detailed in this thesis could be supplemented by real functionality of each block. Especially for blocks like switches and busses, implementing functionality could lead to different results.

Translating the If block. While the If-block has not been part of the blocks detailed in this thesis, it has exactly the features we want to highlight in this section. The if-block (as seen in Figure 40) implements a standard if-else logic.

It has a specifiable number of inputs and outputs, with the inputs being used in if and elseif expressions and the outputs being the outputs of these. It is then connected to Action subsystems, which are run if the expression is true. In our current translation, we translate the if-block as a FUNCTION vertex with variable inputs and outputs.

![Figure 40: If example from [Mat11b]](image-url)
The problem with this approach is, that each output is translated as a part of the graph, even if it isn’t reached. While this doesn’t affect our results in a big way, since those outputs can be run at the same time on different processors, it could still make a difference, especially considering that it would mean that less tasks would have to be scheduled.

Expressions in FUNCTION vertices. To implement this, vertices provide a parameter called expression which can represent any expression used in functional programming. For this it provides operators like CONST() for constants, REF() for referencing edges and ASSIGN() to assign a value to something. As an example an add-block receiving two inputs in1 and in2 and outputting the sum of both to out, resulting in out = in1+in2 would have the following expression: ASSIGN(REF(out), ADD(REF(in1), REF(in2))).

Defining expressions. Blocks in Simulink all have underlying functionality. To implement this in our data flow programs, we have to define such expressions for each block dynamically. For this, various things are needed.

- Standard edge names to be used with the REF operator
- Vertices need to already know their neighbors when created
- Actual functions for each Simulink block

In order to be able to define these expressions, points one and two need to be fulfilled. A vertex needs to know its neighbors to construct the name of its incoming or outgoing edges (i.e. VertexA-VertexB), which is only possible if all edges use the same naming scheme. The biggest problem, and reason why this feature has not been implemented yet is the time-consuming realization of point three as well as some limitations of the Simulink parser library (such as extracting the if-else condition of a port).

A function such as out = in1 + in2 for the Sum block needs to be written for each block provided by Simulink. It would also be needed to be able to translate MATLAB-functions to the expression format, since these can be
used as Simulink blocks as well. Considering the amount of time needed to implement this feature, we have decided to not include it in this thesis.

6.2.2 Precise Complexity Analysis

Another point of interest is a better approximation of a block’s complexity (or even an exact value) hinted at in Section 6.1.1. In order to utilize a better cost model, we need to be able to do one of the following things:

- Execute blocks in Simulink several times and get the mean time it takes to execute the block
- Analyze a block’s function (as defined in Section 6.2.1)
- Get platform-specific approximations by tools.

The first point is very dependent on the hardware it is run on. A Simulink simulation will run at very different speeds depending on both the target system specified as well as the system running the simulation.

Point two is a viable option if the expression format detailed in the previous section is used. Having an expression such as $z = x + y$ means that we can also calculate the complexity of this operation. These expressions have to be defined first and don’t involve the target architecture at all.

Analyzing the Worst Case Execution Time. The third solution would be to use various Worst Case Execution Time (WCET) analysis tools such as aiT by AbsInt [Abs11, FH04] or Chronos [LLMR07]. The WCET is the upper bound $t$ of the execution time of a program. That means that whatever the input, the execution time will never exceed $t$. While these estimates are normally done late in the development process, they can be applied earlier, which enables the developer to make better design decisions [GAEL09].

aiT is a tool specifically for real-time systems. It analyzes a tasks behavior and computes upper bounds for the WCET of a specified system. While this is dependent on the target system, this is not a problem if this analysis can be run before the analyses as specified by this paper are applied. This way a cost model can be developed specifically for each system.
6.3 Conclusion

Embedded Multicores will only continue to grow in importance in the following years as more functions are expected and more computational power is needed with the power consumption as well as the heat generation still being minimized as far as possible. As stated in the beginning of this thesis, the industry will (and in part already does) hit a wall with single-core architectures [Gee05]. The problem with this is the different approach parallel programming brings as well as the need to evaluate the type of architecture (i.e. how many cores to utilize) has to be provided.

Applying the analyses discussed in this thesis, supplemented by the parts that have been classified as future work, can lead to a better implementation of concurrent software for embedded systems, thus easing the transition to multi-core architectures. This thesis lays the groundwork for a set of analyses that can be applied to models constructed with tools such as MATLAB/Simulink in order to make architectural choices in the software development process.

While the translation process for Simulink models only addresses the structural integrity of a model, a functional translation of the model can still be implemented, resulting in an even more accurate description of the system through the use of a fitting cost model. Even with the structural analysis detailed in this thesis, substantiated decisions can be made by utilizing the three metrics detailed in this work. The makespan of an iterative system (as well as its connected metrics speedup and efficiency) tell all about the part of a system that we deemed as important: responsiveness.

As the analysis uses a homogeneous synchronous dataflow graph as its model of computation, the process is not limited to Simulink models and can be adapted for most modeling- as well as synchronous dataflow based languages. Not only will the usage of such analyses result in a more responsive system, providing a higher degree of productivity, it also has the potential to save financial assets during a development process by showing the efficiency of a specific architecture. This way, hardware that fits the needs of the
software can be selected, leading not only to a higher efficiency concerning
the system but also a higher efficiency in the software engineering process.
List of Figures

1  Four Dimensions of this Thesis ............................. 4
2  A Simulink Block ........................................... 6
3  Commonly used Sources and Sinks .......................... 8
4  Mathematical Operation Blocks .............................. 9
5  Delay Blocks ................................................ 9
6  Structural Blocks ........................................... 10
7  Signal Routing Blocks ....................................... 11
8  A Simulink Model ............................................ 12
9  ConQAT Analysis as a Data Flow Graph [FDLHJ10] ....... 14
10 UML Diagram of the Simulink Model representation (State-
    flow not included) ........................................ 15
11 A dataflow graph with four vertices and three edges. .... 16
12 A synchronous data flow with annotated edges [SB09]. .... 17
13 A homogeneous synchronous dataflow graph. ............... 19
14 Example of a schedule (of the HSDFG in Figure 13). ....... 21
15 Workflow for Phase 1 of Vertex generation ................ 30
16 Workflow for Phase 2 of Vertex generation ................ 31
17 Workflow followed while translating lines to edges. ....... 32
18 Workflow while creating the graph object ................... 33
19 Workflow when converting delay blocks to delayed edges ... 33
20 Workflow when unfolding a graph .......................... 34
21 Example of Input and Output Blocks ......................... 36
22 Example of a Constant Block’s Translation ................ 37
23 A model using a Data Type Conversion Block and its translation 38
24 Translation of a Logic Operator ............................ 39
25 Translation of a Relational Operator ........................ 39
26 Translation of a Sum block ................................. 40
27 Translation of a Product block .............................. 41
28 Translation of a Gain block ................................. 41
29 Translation of a MinMax block .............................. 42
30 Translation of Delay Blocks (Zero-Order Hold omitted) .... 42

70
List of Tables

1. Four classes of scheduling [Lee89] ..................................... 21
2. New variables in the Vertex class .................................... 30
3. Twenty most-used Simulink Blocks in a real-life industry example ................................................................. 35
4. Minimum makespan results for 2- and 4-core architectures. .............................................................. 57
5. Minimum makespan results for 8- and 16-core architectures. .............................................................. 57
6. Number of vertices and edges per layer. ...................................... 57
References


A Appendix

Evaluation Result Tables with Execution Times

This section includes the results of the analyses conducted on the Simulink model provided. Execution times of the scheduling algorithm are given where applicable (i.e. Layer 1 and single-core schedules only show the iterative makespan, so running the analysis for one of them sufficed)

<table>
<thead>
<tr>
<th>Layer</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Layer 2</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>Layer 3</td>
<td>63</td>
<td>94</td>
</tr>
<tr>
<td>Layer 4</td>
<td>164</td>
<td>276</td>
</tr>
</tbody>
</table>

Table 7: Number of vertices and edges per layer.

<table>
<thead>
<tr>
<th>Depth</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 3</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 4</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
</tbody>
</table>

Table 8: Minimum makespan results for a single-core architecture (iterative makespan).
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230 (0.45s)</td>
<td>230 (0.52s)</td>
<td>306 (0.50s)</td>
<td>327 (0.22s)</td>
</tr>
<tr>
<td>Depth 3</td>
<td>165 (193.68s)</td>
<td>165 (222.27s)</td>
<td>165 (275.57s)</td>
<td>165 (55.496s)</td>
</tr>
<tr>
<td>Depth 4</td>
<td>165 (46.9h)</td>
<td>165 (34.6h)</td>
<td>165 (32.9h)</td>
<td>165 (46.5h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230 (0.46s)</td>
<td>230 (0.40s)</td>
<td>306 (0.29s)</td>
<td>327 (0.19s)</td>
</tr>
<tr>
<td>Depth 3</td>
<td>82 (254.97s)</td>
<td>82 (918.61s)</td>
<td>90 (40.88s)</td>
<td>90 (13.642s)</td>
</tr>
<tr>
<td>Depth 4</td>
<td>82 (23.9h)</td>
<td>84 (21.8h)</td>
<td>90 (16.2h)</td>
<td>90 (18.0h)</td>
</tr>
</tbody>
</table>

Table 9: Minimum makespan results and analysis execution time for 2- and 4-core architectures.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230 (0.484s)</td>
<td>230 (0.491s)</td>
<td>306 (0.339s)</td>
<td>327 (0.19s)</td>
</tr>
<tr>
<td>Depth 3</td>
<td>77 (14.184s)</td>
<td>77 (17.508s)</td>
<td>91 (8.955s)</td>
<td>91 (6.279s)</td>
</tr>
<tr>
<td>Depth 4</td>
<td>51 (13.0h)</td>
<td>51 (14.7h)</td>
<td>90 (10.9h)</td>
<td>90 (11.6h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>Depth 2</td>
<td>230 (0.348s)</td>
<td>230 (0.381s)</td>
<td>306 (0.282s)</td>
<td>327 (0.216s)</td>
</tr>
<tr>
<td>Depth 3</td>
<td>77 (9.651s)</td>
<td>77 (7.98s)</td>
<td>91 (9.144s)</td>
<td>91 (5.941s)</td>
</tr>
<tr>
<td>Depth 4</td>
<td>51 (5.8h)</td>
<td>51 (5.1h)</td>
<td>90 (3.9h)</td>
<td>90 (2.4h)</td>
</tr>
</tbody>
</table>

Table 10: Minimum makespan results and analysis execution time for 8- and 16-core architectures.
Line-Graphs for the Evaluation

This section contains graphs of the evaluation results from various views.

![Figure 41: Message cost/makespan chart for a dual-core architecture.](image)

Layer	
  1	
   327	
   327	
   327	
   327	

Layer	
  2	
   230	
   230	
   306	
   327	

Layer	
  3	
   165	
   165	
   165	
   165	

Layer	
  4	
   165	
   165	
   165	
   165
Figure 42: Message cost/makespan chart for a quad-core architecture.

Figure 43: Message cost/makespan chart for a octa-core architecture.
Figure 44: Message cost/makespan chart for a 16-core architecture.

Figure 45: No. of cores/makespan chart for a message cost of 1.
Figure 46: No. of cores/makespan chart for a message cost of 10.

Figure 47: No. of cores/makespan chart for a message cost of 100.
Figure 48: No. of cores/makespan chart for a message cost of 1000.

Figure 49: Message cost/makespan chart for a translation depth of 2.
Figure 50: Message cost/makespan chart for a translation depth of 3.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>2</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>82</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>77</td>
<td>77</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>16</td>
<td>77</td>
<td>77</td>
<td>91</td>
<td>91</td>
</tr>
</tbody>
</table>

Figure 51: Message cost/makespan chart for a translation depth of 4.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>327</td>
<td>327</td>
<td>327</td>
<td>327</td>
</tr>
<tr>
<td>2</td>
<td>165</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>84</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>51</td>
<td>51</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>16</td>
<td>51</td>
<td>51</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>