Modellbasierte Konfiguration einer grobkörnigen rekonfigurierbaren Architektur
Model-Based Configuration of a Coarse-Grained Reconfigurable Architecture

Jens Frömmer, Dr. Nico Bannow, Axel Aue, Prof. Dr. Christoph Grimm, Prof. Dr. Klaus Schneider
Robert Bosch GmbH, Stuttgart, Germany, {jens.froemmer, nico.bannow, axel.aue}@de.bosch.com
TU Kaiserslautern, Kaiserslautern, Germany, {froemmer, grimm, schneider}@cs.uni-kl.de

Kurzfassung


Abstract

Domain specific coarse-grained reconfigurable architectures offer a performance and efficiency close to hardwired implementations when accelerating multiple selected data intensive algorithms. However, this requires the transfer of algorithms onto the coarse-grained reconfigurable architecture to lead to an as high as possible utilization. This paper introduces a modeling framework for a model-based configuration workflow enabling algorithm experts and hardware experts to collaboratively and efficiently transfer algorithms onto a coarse-grained reconfigurable architecture. The concept was validated by the implementation and the testing of respective prototypes.

1 Introduction

In the automotive industry, algorithms for data based modeling, advanced control theory, advanced signal processing, and physical modeling are expected to run on the next generation of engine control unit (ECU) microcontrollers. Hence, rising performance requirements have to be addressed by these microcontrollers despite the high cost constraints in the ECU market. A novel coarse-grained reconfigurable architecture (CGRA), called Data Flow Architecture (DFA), has been developed at Bosch in accordance to the requirements and constraints of the automotive field. This paper contributes a model-based approach to transfer algorithms onto the DFA:

1. A Simulink-based modeling framework enables the modeling of algorithms with the DFA specific processing elements, called base blocks, and simulates the execution of the modeled algorithm on the DFA. The modeling framework offers different levels of abstraction from the configuration options of the DFA via a custom library and masks. Hence, DFA experts can access all functional configuration options available in the DFA, while algorithm experts without extensive knowledge of the DFA can rely on abstracted configuration options or predefined elements.

2. A configuration workflow imports the DFA-based models of algorithms as well as models of a DFA hardware accelerator to enable the mapping of algorithms onto the DFA and the generation of a respective DFA-configuration. The output of the code generator comprises a SystemC test bench and a VHDL test bench. The meta-models and the code generator are based on the Eclipse Modeling Framework (EMF) and the Eclipse Xpand.

1.1 Transferring Algorithms onto a CGRA

Creating highly optimized configurations manually at the register-level is time-consuming, error-prone, and requires a deep understanding of the targeted CGRA as well as the selected algorithm. Hence, a framework or tooling is required to enable DFA experts as well as non-DFA experts to transfer algorithms onto the DFA - each on their own and in collaboration with each other. Any framework or tooling has to allow for the efficient utilization of the target architecture and the modeling as well as the verification of algorithms before implementing them in hardware [1]. Earlier compilers, such as DRESC [2] and further ap-
proaches [3], enable the use of high level languages to program a CGRA, but usually assume a mesh-like interconnect, homogeneous processing elements, or other properties that do not apply to the DFA. More recent research improves upon different aspects, e.g. support of route sharing [4], register-awareness [5], or memory-awareness [6]. The Split-Push Kernel Mapping (SPKM) [7] explicitly extends the range of supported CGRAs. As finding the optimal mapping of an algorithm onto a CGRA is in general NP-complete [8], compilers often rely on heuristics like the SPKM and, thus, achieve only a limited utilization of the targeted CGRA. As the DFA features runtime-reconfigurable and heterogeneous base blocks that are connected via a sparse crossbar and offer complex configuration options, existing compiler or mapping techniques cannot simply be applied to the DFA.

The introduction of a specific instruction set architecture (ISA) similar to the stream-dataflow ISA [9], may lead to a high utilization, but typically requires deep knowledge of the CGRA. Hence, a specific ISA presents an initial hurdle for the widespread acceptance of the CGRA. Furthermore, a new ISA may be difficult to integrate into established workflows and tools, especially in case of the model-driven automotive software engineering.

The application in the automotive field demands for the acceleration of selected algorithms and a high performance per size ratio. An effective usage of the DFA resources has a higher priority than the transfer of arbitrary algorithms. Therefore, a model-based approach offering a graphical user interface is preferred. Compared to a compiler, a model-based workflow requires manual work for each algorithm, but allows for an interactive mapping onto the CGRA that is not limited by a fixed compiler implementation. A modeling framework that initially abstracts from the CGRA promises better accessibility than a specific ISA. Little work has been published on structural and graphical approaches to program a CGRA. One example is MorphoSys [10], which offers a graphical user interface (GUI) called mView to generate the application context file and to simulate the processing elements.

1.2 Background and Related Work

The modeling framework as well as the configuration workflow are based on existing tools and frameworks in order to integrate well into existing software engineering workflows and to speed up the development of the respective prototypes.

1.2.1 Data Flow Architecture

The DFA is an architecture for flexible hardware accelerators. A DFA hardware accelerator is to be integrated into the microcontroller, mapped into the global address space to be accessible by the masters, and has a master interface for direct memory access capabilities with a high bandwidth. The base blocks of the DFA are connected via a sparse crossbar and implement control, load, store, and mathematical operations. Additionally, other functionalities, such as a software core, can be integrated as base blocks. The base blocks may only write to their successors, but not read from their predecessors. Only the control, load, and store base blocks may access the local SRAM or global memory. Each base block offers configuration options to specify its exact operation, e.g. to define the memory access pattern of a load block. Hence, a DFA configuration comprises configurations of the global registers, base blocks, SRAM, and eventually debug features. From an algorithmic perspective, a DFA configuration describes a data flow: The configuration of the load and store base blocks determines which data flows in and out of the operations as well as the order of the data. The configuration of the operational base blocks like the multiply-accumulate (MAC) base block determines the types and the order of the operations. The execution of an operation depends on the availability of the required operands. The emulation of data flow graphs gives the DFA its name, while it is not closely related to classical data flow architectures, such as the Manchester Prototype Dataflow Computer [11]. Figure 1 visualizes the DFA-specific computation of the dot product. The design space of the DFA especially includes the structure and the available connections of the interconnect as well as the types, the quantities, and the features of the base blocks.

As a detailed description of the DFA would go well beyond the scope of this paper and is not required for the introduction of the configuration workflow, it is published separately.

During the introduction, the term base block (BB) refers to a BB of the DFA. Starting from Section 2, these are called hardware base blocks (HWBBs) to differentiate them from the virtual base blocks (VBBs) of the modeling framework. These VBBs are the functional counterparts of the HWBBs.

1.2.2 Simulink

Simulink offers a broad library of blocks, which can be used to graphically model dynamic systems in block diagrams. Those blocks have input and output ports, can be parameterized, and allow for the modeling of hierarchical structures. The mathematics of a Simulink block are expressed in Table 1. Simulink blocks are connected to each other via signals linking output to input ports. The output values may change over time and signals can be of a complex as well as multi-dimensional data types. Continuous
and discrete states are both supported. Having modeled a dynamic system in such a block diagram, it can be simulated to analyze its behavior using fixed- or variable-step solvers. As Simulink is an established software engineering tool in the automotive industry including Bosch and a graphical data flow modeling language [12], it is the starting point of the development of the modeling framework.

1.2.2 Simulation

The simulation of a Simulink model is executed in several phases [13]: model compilation, link phase, and simulation loop phase. During the first phase (model compilation), the model is converted to an executable form. To that end, block parameters are evaluated, signal attributes as well as block execution order and sample times are determined, model hierarchy is flattened, and blocks are optimized. In the second phase (link phase), the required memory is allocated and the method execution order is determined. Finally, an initialization is performed once in the simulation loop phase, before states and outputs are computed at each time step as visualized in Figure 2. The time steps are determined in conformance to the defined simulation start and stop time as well as the block sample times. The computation of states and outputs is performed for each block in the previously scheduled execution order.

Table 1 Function parameters and equations that represent a Simulink block. Output, derivative, and update calculation depend on the current simulation time, states, and inputs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>simulation time</td>
</tr>
<tr>
<td>$x$</td>
<td>continuous states</td>
</tr>
<tr>
<td>$x_c$</td>
<td>discrete states</td>
</tr>
<tr>
<td>$y$</td>
<td>outputs</td>
</tr>
<tr>
<td>$x_d$</td>
<td>discrete states at time step $k$</td>
</tr>
<tr>
<td>$x_{d,k+1}$</td>
<td>derivatives</td>
</tr>
<tr>
<td>$f_0(t,x,u)$</td>
<td>update</td>
</tr>
<tr>
<td>$f_d(t,x,u)$</td>
<td>update</td>
</tr>
</tbody>
</table>

The time-based simulation of Simulink differs from traditional hardware simulation kernels like the ones of SystemC [14] and VHDL [15], which use discrete event simulation. In case of Simulink, the time always progresses after the computations of the states and the outputs. Thus, there are no equivalents to the delta cycles of hardware simulation kernels, which do not induce a progress in time. Additionally, high level communication channels, such as blocking, are not supported by Simulink. However, using fixed sample times as well as discrete states allows for a cycle-like behavior. The missing delta cycles and high level communication channels can partly be compensated by exploiting the state and output computations of user-defined Simulink blocks.

1.2.2.2 S-Functions

Simulink supports different kinds of user-defined blocks: subsystem blocks, Matlab function blocks, Matlab system blocks, S-Function blocks, and masked blocks. The functionality of an S-Function (block) is defined by Matlab, Fortran, C or C++ code [16]. A mask is used to parametrize a block. The Matlab execution engine can automatically load and execute S-Functions after their compilation. S-Functions may contain continuous, discrete, or hybrid systems.

S-Functions implement callback methods that are called from the simulation engine of Simulink. There are methods for each stage of the simulation loop phase. The initialization, the output calculation, and the update of the discrete states are the most relevant phases for the modeling framework.

1.2.3 Eclipse

Eclipse describes itself [17] as “a mature, scalable and commercially-friendly environment for open source software collaboration and innovation.” The top-level Eclipse project contains a framework for tool integration as well as a Java development environment, which is built using the framework for tool integration. Due to its platform-based approach, Eclipse is easily extensible. According to a recent survey [18], Eclipse is one of the most popular development environments. This and the EMF are the main reasons, why Eclipse was chosen as a basis for the configuration workflow.

1.2.3.1 Eclipse Modeling Framework

The EMF provides a meta-model called Ecore as well as a number of code generation options [19]. Its objective is to increase the productivity when developing a tool manipulating a structured data model. Therefore, it allows for the modeling of a data meta-model on the basis of the Ecore meta-model. Given such a data meta-model, Java code can be generated in order to:

- create, directly access, and manipulate models,
- access models using adapter classes, and
- edit models via a generated editor.

The EMF supports model specifications described in the XML Metadata Interchange (XMI) format, which especially includes support for the eXtensible Markup Language (XML) and the Unified Modeling Language (UML). Furthermore, models can be defined via Java interfaces. The modeling of meta-models via Ecore as well as the code generation options, especially the generated editors and the direct access to the models, cover many features required
by the configuration workflow. Therefore and because of
the code generation options, the configuration workflow
builds upon it.

1.2.4 Code generation

The Eclipse Modeling Project includes three template-
based model-to-text transformation projects supporting the
EMF: Acceleo, Jet, and Xpand. According to Syriani et
al. [20] template-based code generation may be described
“as a synthesis technique that uses templates in order to
produce a textual artifact, such as source code, called the
output.” A template consists of a static and a dynamic part.
Text fragments that directly appear in the output are con-
sidered as static parts, whereas dynamic parts consist of
meta-code. This meta-code is executed by the template en-
gine, thus computing the output based on run-time input.
The meta-code also describes what kind of run-time input
is expected. Usually, it is a model or a part of a model.
While Acceleo, Jet and Xpand are all viable solutions for
generating the configuration and test benches, an earlier
comparison [21] favors Xpand for a number of reasons,
such as Xpand being the most powerful of the three or
missing domain-specific support in the case of Jet. Other
than Jet and Acceleo, Xpand offers straightforward means
to pass an arbitrary number of models to a template. As
multiple models, e.g., a mapping model and a data flow
model, need to be evaluated in parallel for the generation
of a DFA configuration, this is a required feature for the
configuration workflow. Hence, the code generation tem-
plates of the configuration workflow use Xpand.

1.3 Outline

The remainder of this paper is structured as follows:
Firstly, Section 2 presents the concept of the Simulink
modeling framework and the configuration workflow. Sec-
ondly, Section 3 elaborates how these concepts are real-
ized based on the above introduced tools and frameworks.
Thirdly, Section 4 briefly reviews the chosen tools and
frameworks before introducing a proof of concept by test-
ing the prototypes. Finally, Section 5 summarizes and dis-
discusses the findings of this paper.

2 Concept and Requirements

While being separated, the modeling framework and the
configuration workflow must work well together.

2.1 Modeling Framework

As explained before, the HWBBs resemble data flow op-
erations that can be connected to each other. For each
HWBB, a counterpart is implemented in Simulink as an
S-Function that offers the same behavior and the same con-
figuration options, but only at the functional level. These
counterparts are the VBBs. The functionally identical con-
figuration options between the HWBBs and the VBBs sim-
plify the mapping framework and the code generation of
the configuration workflow.

A major challenge regarding the Simulink modeling
framework is the realization of a behavior similar to real
hardware. The VBBs implemented as S-Functions must:
• delay their output by one time step,
• block their execution in case of unfulfilled data depen-
dencies,
• block their data output in case of a subsequent block-
ing VBB,
• be independent of the execution order, and
• resemble the functionality as well as the configuration
options of the HWBBs.

Aligned to the simulation loop of Simulink, a rather sim-
ple solution to delay the output by one time step is to
perform the computation of new values during the update
phase. The computed values are stored in internal states
and copied to the output during the next time step. In case
of unavailable inputs, a base block computes no values and
signalizes invalid outputs. As the VBBs are connected to
each other and may have multiple in- and outputs, the fol-
lowing case is possible: The successor of a VBB cannot
continue its computation because another required input is
unavailable. Thus, the prior VBB has to wait for its suc-
cessor to continue or its outputs need to be buffered. Of
course, the VBBs must produce the same output values in
the same order at identical time steps independently of their
execution order.

2.2 Configuration Workflow

Given an algorithm and a DFA hardware accelerator, a
configuration of the DFA hardware accelerator that results
in the computation of the algorithm can be derived as de-
picted in Figure 3: The first step is to create and import a

![Figure 3](image-url)

Figure 3 The configuration workflow comprises all nec-
essary steps to transfer an algorithm to the DFA.

corresponding Simulink model using the modeling frame-
work. In parallel, a representation of the DFA hardware
accelerator must specify the relevant characteristics: the
structure and available connections of the interconnect as
well as the quantities, types, and features of the HWBBs.
These information are available in IP-XACT, which is an XML format for language and vendor-neutral descriptions of electronic circuit designs [22]. The availability of the data flow model and the hardware model enables the mapping of VBBs onto HWBBs via references. A variety of mapping options may be explored, since several HWBBs may match a VBB. Furthermore, the HWBBs offer features for resource-sharing: multiple execution units as well as register sets per HWBB and multiple transfer cycles per port. Finally, the code generator can create a VHDL as well as a SystemC test bench including the configuration of the DFA hardware accelerator by evaluating the data flow model, the hardware model, and the mapping model. Importing and evaluating the hardware model enables the generation of the respective architecture in VHDL and SystemC.

2.2.1 Importing a Simulink Model
Importing the Simulink models into custom data flow models allows for a clear interface separating the configuration workflow from the modeling framework. The separation results in a configuration workflow that is independent of future changes to Simulink, especially changes to the structure of the Simulink models. Furthermore, it enables the use of diverse modeling frameworks. By using an Ecore meta-model for the data flow model, the EMF can generate the Java classes to directly load, manipulate, and store the data flow models including a model-editor.

An importer can traverse and evaluate the Simulink model using the XML Path Language (XPath), since it consists of compressed XML files. The meta-model of those XML files must be reverse engineered and implemented manually based on exemplary models, since there is no XML schema available for the Simulink XML file format.

2.2.2 Mapping Framework
As the data flow model and the hardware model are based on the EMF, there are two options for the mapping model: 1. combining data flow and hardware model with the addition of the mapping information or 2. referencing VBBs and HWBBs using the respective identifiers. The first option would lead to a single model that contains all information required for the code generation. As this model would also contain redundant information, option two is preferred resulting in three strictly separated models.

2.2.3 Code Generation
The use of Xpand templates combined with supportive Java classes enables the evaluation of multiple models in parallel. The Xpand templates can be separated into the generation of the DFA configuration including test benches in VHDL and SystemC as well as the generation of the architectural description in VHDL and SystemC.

3 Implementation
Prototypes of the modeling framework and configuration workflow were implemented to provide a proof of concept and a starting point for the future tool development regarding the DFA.

3.1 Modeling Framework
Figure 4 visualizes the modeling framework: The core of the modeling framework, which is based on Simulink, is a custom library of VBBs that are implemented as S-Functions. These VBBs equal HWBBs in their functionality and their configuration options. Additionally, there are different variants of these VBBs available that offer predefined configurations and simplified masks to match certain operations, such as a multiplication or an subtraction. All VBBs use a fixed sample time to emulate clock cycles. An algorithm modeled with the VBBs can be simulated to ensure that it behaves as intended. Predefined models of algorithms are available in the library as well. The communication of the VBBs relies on custom bus objects that are stored in a data dictionary. A data dictionary is a persistent storage that may store bus objects, parameters, and other data defining the behavior of a model. A data dictionary can be linked to a Simulink model in order to allow for access to the stored data. A bus object may consist of multiple signals.

Figure 4 The components of the modeling framework.

In general, methods, macros, and constants that realize shared functionalities are implemented in respective header files to avoid redundancies and to allow for reuse and extensibility. E.g. Simulink, callback, bus, and parameter related functionalities are encapsulated in header files.

3.1.1 Types of Virtual Base Blocks
The VBBs may be categorized as: accessing memory, performing mathematical operations, and controlling the data flow. The local SRAM of the DFA is emulated via memory blocks that allocate a one dimensional array, provide a pointer for read-access, and store incoming data at the address attached to the data. Test bench blocks are memory blocks that initialize this one dimensional array with algorithm-specific test data and check incoming data against the expected results. The memory blocks may only be connected to the memory accessing VBBs, which are the load and the store VBBs. The load and the store VBBs allow for the access of the data via their address generation that can compute complex memory access patterns. The memory accessing VBBs are usually connected to VBBs...
performing mathematical operations like addition, subtraction, multiplication, accumulation, exponential function, and more. Additionally, there are VBBs that perform special operations, which either control the data flow or are specific to the modeling framework. The latter cannot be found in an actual DFA hardware accelerator. A simple example is the DataToConfig VBB that converts a DATABUS into a CONFIGBUS to reconfigure a subsequent VBB on the basis of a computed value contained in the DATABUS.

### 3.1.2 Bus Objects

There are five different types of bus objects: ACCUBUS, CONFIGBUS, DATABUS, READBUS, and WRITEBUS. VBBs that accumulate values use the ACCUBUS to internally store an accumulated value. The CONFIGBUS contains one or multiple values and targets, which specify configuration parameters. The DATABUS transports data, which was loaded from the memory or computed. Only the memory blocks as well as the memory accessing VBBs use the READBUS and the WRITEBUS. The READBUS may contain a pointer to a one dimensional array, while the WRITEBUS may contain a value and an address. All of the bus objects include valid flags. The DATABUS and the CONFIGBUS additionally include information about the loop level, e.g. to control when to output and reset an accumulated value.

### 3.1.3 Block Synchronization

When starting the simulation or in case of data dependencies, a mechanism is required to either buffer an arbitrary amount of data in between the VBBs or to postpone further computations until the next VBB is ready. E.g. an add VBB, which computes the sum of two inputs $A$ and $B$, is connected to a second add VBB. The input $D$ of the second add VBB is unavailable. Consequently, the second add VBB cannot process $C$ to compute $E$ as highlighted in Figure 5.

![Figure 5](image)

Figure 5 A simple example of a data flow that requires a synchronization method such as blocking or a buffer.

As HWBBs would block in such a scenario, two approaches implementing a synchronization have been evaluated: feedback-connections and a callback functionality. Additional feedback-connections from a subsequent VBB to its previous VBB allow a VBB to accept or decline a provided input. However, for each output port an additional input port is required. This results in a very confusing connectivity structure even in case of small models. The callback functionality is not visible by connections in the model while providing the same functionality: The second add VBB calls a function of the first add VBB via a callback pointer, setting a flag which signals that the subsequent VBB is not ready for new inputs or, in other words, has not consumed the provided input yet. Therefore, the first add VBB stops its computation and waits. As the circumstances require, the first add VBB calls also the callback function of the previous VBB in order to block the computation of the previous VBB. Whenever $D$ becomes available, the second add VBB continues its computation and calls the callback function of the first add VBB. This time, it sets a flag signaling that the subsequent VBB is ready for new values and has consumed the provided input. In addition to the flag, the callback function requires any VBB to send a port index. Thereby and by having separate flags for each output port, a VBB can wait for all subsequent VBBs to become ready. All VBBs contain and utilize the callback functionality, but not the memory blocks. The memory blocks do not require any synchronization, since memory access conflicts etc. are not modeled.

### 3.1.4 Configuration Parameters

Different masks per VBB allow for different levels of abstraction and provide the actual S-Functions with the configuration parameters. Figure 6 presents the mask of a MAC VBB including a few configuration options related to the accumulator of the MAC VBB: E.g. the internal source of the accumulated value may either be the result of the multiplication or the result of the addition. While the library of the prototype includes only a limited number of masks or abstraction levels, future extensions are simple and fast due to the mask editor of Simulink. Matlab scripts enhance the masks via the callback and the initialization methods of the masks.

![Figure 6](image)

Figure 6 A screenshot of the mask of a MAC VBB in Simulink.

### 3.1.5 Predefined Algorithms

The predefined algorithms are subsystem blocks with a mask that contains an initialization script. This script generates, connects, and configures the underlying VBBs in accordance to the algorithm-specific configuration parameters of the mask. E.g. the mask of a matrix multiplication block requires the sizes of the input matrices. Using the
provided sizes, the initialization script computes the dynamic configuration values, such as the base addresses of the individual matrices. The user is able to explore the resulting VBBs and their configurations by opening the generated subsystem. The predefined algorithms simplify not only the simulation and testing of various input dimensions, but abstract also from the configuration options of the underlying VBBs.

3.1.6 Virtual Base Block

The core functionalities of the VBB that are not related to the simulation engine of Simulink, such as the evaluation and generation of loop level information, mathematical operations, and the address generation, were implemented as independent C++ classes to enable reuse. All classes inherit from a base class that defines generic attributes and methods as well as a virtual operation method.

3.1.6.1 Initialization Phase

The initialization sets the sample time, declares the port types, and the number of ports per type, checks the parameters, and allocates memory for the internal states. Many of the initialization values are derived from the configuration parameters, e.g., the number of ports per type. Additionally, the class that corresponds to the VBB is instantiated and provided with the respective configuration parameters. Afterwards, the simulation loop starts. For every time step during the loop phase the output methods of all VBBs are invoked, followed by all update methods.

3.1.6.2 Update Phase

First of all, the inputs are evaluated and copied to internal states, if these are not blocked by previous operands. Depending on the configuration, the execution of a VBB may consume the operands. In case of CONFIGBUS inputs, the contained reconfiguration values are applied, if the targeted states may be overwritten. Afterwards, the update methods of the VBBs perform the actual operation to compute the next output values and states: If all operands that are required for the configured operation are available, the operands are passed on to the instance of the class that corresponds to the VBB. The operation method of the same instance is invoked and, if the output is enabled by the loop level, the result of the operation is assigned to the next update of the outputs. Finally, the VBB informs its previous VBBs, which provided the inputs, whether it is ready for new inputs via the callback functionality explained above.

3.1.6.3 Output Phase

The output methods of the VBBs assign the updates that were computed during the update phase of the previous time step to the respective states and outputs. E.g., the flags indicating whether subsequent VBBs are ready for new inputs are not updated directly via the callback functionality. Instead, the updated flags are stored as copies before their assignment during the output phase. This ensures a consistent behavior that is independent of the execution order of the VBBs.

3.2 Configuration Workflow

The IP-XACT models of the DFA hardware architecture were defined outside the scope of the configuration workflow. However, the IP-XACT XML schemes were imported in EMF and the respective Java code to load, store, and manipulate IP-XACT models including a simple editor was generated. Therefore, those IP-XACT models can directly be passed to the code generator.

3.3 Simulink XML

The Simulink model description in XML can either be obtained by decompressing the Simulink model file or by exporting the Simulink model to an XML file via the save_system command. The latter may not be available in all versions of Simulink. The XML file of a Simulink model is structured as follows: Apart from additional model information, such as the editor and simulation settings, the description of the model itself is encompassed by the <System> tag. It comprises <Block> and <Line> tags, which describe the Simulink blocks and their connections. The attributes of the <Block> tag define the name and the type of the block. The configuration parameters of the mask of the block are within <P> tags encompassed by <Object> and <Array> tags. Connections are realized via references to blocks and ports within the <Line> tags.

3.4 Data Flow Meta-Model

Naturally, the description of a Simulink model is well suited to describe a data flow. Hence, the data flow meta-model of the configuration workflow is quite similar to the structure within the <System> tag of a Simulink model. The main difference is the introduction of specific block types corresponding to the VBBs. The respective Java code to directly load, store, and manipulate the data flow models as well as a simple editor was generated via the EMF.

3.5 Simulink Model Importer

The Simulink model importer was implemented in Java and traverses a Simulink XML file using XPath. In parallel, the importer creates and fills a new data flow model. The information extracted from the Simulink model comprise the names, the types, the parameters, and the connections of the VBBs and memory blocks.

3.6 Mapping Meta-Model

The mapping meta-model was implemented using the EMF and its code generation options, too. Each entry of a mapping model references exactly one VBB and one HWBB. Supplementary information specifies the execution unit and the register set of the HWBB.

3.7 Code Generation

The code generator shown in Figure 7 comprises the GUI, the generator, and the utility classes implemented in Java as well as the configuration and test bench templates, the base block templates, and the architecture templates im-
implemented in Xpand. The GUI collects the required models: data flow model, hardware model, and mapping model. Furthermore, the GUI passes the models on to the generator and invokes it. The generator starts the code generation of Xpand using the configuration and test bench templates as well as the architecture templates. Each target, currently VHDL and SystemC, requires an explicit configuration or test bench template as well as an architecture template. Of course, the architecture templates are required only, in case the architecture is not already described in the targeted SystemC or VHDL model. If the architecture is already described in VHDL or SystemC, the identifiers of the HWBBs in SystemC and VHDL must comply with the identifiers of the HWBBs in IP-XACT.

### 3.7.1 Templates

The configuration and test bench templates contain static initializations, e.g. providing test data, and invoke the generic base block configuration templates by traversing all entries of the mapping model. There is one base block configuration template for each HWBB type. The base block configuration templates require the data flow model, the hardware model, and the mapping model as parameters. In a base block configuration template, the respective configuration parameters of the VBB are extracted from the data flow model and used to generate address-data tuples that configure a HWBB. Listing 1 shows the template for an address-data tuple, where `ms_system` refers to the data flow model, `block` to the VBB, and `getParam` to a method of the utility classes. The latter returns the configuration value of a specified configuration parameter of the VBB.

A more complex example is the specification of the target address of a HWBB: The subsequent VBBs of a VBB are found in the data flow model, their HWBB counterparts are identified in the mapping model, and the respective addresses of the HWBBs are derived from the hardware model. As queries that require the parallel traversal of multiple models are difficult or even impossible to realize directly in the Xpand templates, the utility classes extend the templates.

```plaintext
Listing 1

i_BB_Ex_fAddSub
[<<split>>][<<bb>>]
[regId + BB_EX_ADDSUB__REG_ID__OPERATION] =
<<getParam(block, "PARAM_OPERATION_ADDSUB", ms_system)>>;
```

### 3.7.2 Utility Classes

The utility classes implement several methods, which may be seen as queries. These queries often traverse the same parts of a model multiple times. In order to reduce the runtime, the utility classes set up data structures holding the respective model information, which was gained during the first traversal. E.g. the `getParam` query must traverse all configuration parameters of a VBB to find and return a single configuration value. By creating a hash map that is initially filled with all parameters, the parameters of the VBB need to be traversed only once. The utility classes for the base block configurations and the architecture are independent of each other.

### 4 Results

In order to prepare for future work on the modeling framework and the configuration workflow, the succeeding subsection briefly reviews the utilized frameworks and tools before the proof of concept.

#### 4.1 Benefits and Limitations of the chosen Frameworks and Tools

The use of Simulink, Eclipse, EMF, and Xpand enabled the development of the prototypes for the modeling framework and the configuration workflow in a viable amount of time. Beside the initially elaborated reasons for the use of these tools and frameworks, additional benefits and limitations became apparent.

##### 4.1.1 Simulink

When extending Simulink in the foreseen ways, the functionality and the tools of Simulink, such as the simulation data inspector, are applicable. Hence, developers who are familiar with Simulink need only to understand the functionality of the VBBs, but are not required to learn a new tool. The simulation data inspector is well suited to analyze the simulation behavior of a modeled algorithm. As the VBBs are S-Functions, they can be combined with other Simulink blocks.

Apart from these benefits, several limitations became apparent during the implementation of the prototype:
• the block masks offer only limited customization and scripting options,
• all methods of a C(++) S-Function are static,
• bus objects have to be created in advance and made available to any model,
• the data types are insufficient, and
• there is no XML schema for Simulink models available.

These limitations are no critique, but the result of using Simulink for a (simplified) simulation of digital hardware, which Simulink was not designed for. However, most of the limitations were overcome or can be overcome by future work.

4.2 Proof of Concept

In order to validate the proposed modeling framework and the configuration workflow, synthetic tests as well as a full run-through were performed on the basis of the implemented prototypes. The test and development environment is listed in Table 2.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Windows 10, 64 bit</td>
</tr>
<tr>
<td>Matlab, Simulink</td>
<td>R2016b</td>
</tr>
<tr>
<td>C(++) MEX compiler</td>
<td>MinGW GCC 4.9.2 from TDM</td>
</tr>
<tr>
<td>Java</td>
<td>1.8 (JRE)</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Neon.3 Release (4.6.3)</td>
</tr>
<tr>
<td>EMF</td>
<td>2.12.0.v20160526-0356</td>
</tr>
<tr>
<td>Xpand/Xtend</td>
<td>2.2.0.v201605260315</td>
</tr>
</tbody>
</table>

Table 2 The software versions that were used to implement and test the prototypes of the modeling framework and the configuration workflow.

4.2.1 Synthetic Tests

Various synthetic tests were performed throughout and after the development of the modeling framework to validate the simulation behavior. Especially, the callback functionality as well as the exploitation of the update and the output phase to separate the computation of new outputs and states from their actual assignment are critical features of the modeling framework. These features must not only work together for one VBB, but also in case of several communicating VBBs. In order to test these features, multiple models were set up as test benches and run with different execution orders of the VBBs. Figure 8 illustrates a Simulink model that was used to exercise stress tests regarding the behavior of the VBBs: Two data paths with different pipeline lengths end in a single VBB. The model was simulated with varying execution orders specified via the block properties. The assigned order can be seen at the top on the right of the blocks. In addition to the debug output, the outputs of the individual blocks were logged via the data inspector of Simulink. All simulations yielded the exact same outputs confirming that the behavior of the VBBs is insensitive to the block execution order. Additionally, models covering specific scenarios like two synchronized sub-algorithms and models testing individual base block features like the computation of complex memory access patterns were simulated. The results were checked against the expected behavior of the DFA. All synthetic tests were successful showing that the models created via the proposed modeling framework behave on a functional level as the DFA does.

4.2.2 Modeled Algorithms

Algorithms ranging from a simple matrix multiplication to complex Bosch-internal algorithms with series relevance, such as the gaussian process [23], were successfully modeled and simulated. Besides, these algorithms were added as library elements with respective masks and Matlab scripts, which create, connect, and configure the underlying VBBs. An example of a matrix multiplication model can be seen in Figure 9: The model computes \( A \times B \) where \( A \) and \( B \) are matrices. A memory block holds the input matrices and the expected result matrix. The load and store VBBs are connected to the memory block to read the input matrices and write the result matrix. The first load VBB reads the input matrix \( B \) column-wise, while the second load VBB reads the input matrix \( A \) row-wise. The elements of the columns and the rows are sent to the subsequent MAC VBB. The MAC VBB multiplies the incoming values and accumulates the results. The attached loop level information indicates the end of a column or row. After each row or column, the MAC VBB outputs the accumu-
Testbench for \((n \times m) \times (n \times l)\) matrix multiplication

**Figure 9** A matrix multiplication subsystem that was generated via a masked library element.

lated value to the store VBB and resets it to zero. The values sent by the second load VBB as well as the valid flags of the outputs of the MAC VBB are depicted in **Figure 10**: After one initialization cycle or time step, the load VBB outputs a new value at each cycle. After a row, which contains in the given example the four elements \([1, 2, 3, 4]\), was sent by the load VBB to the MAC VBB, the MAC VBB outputs the accumulated value with the valid flag set to true.

4.2.3 Full Run-Through

The algorithms mentioned above were not only successfully modeled and simulated in Simulink, but also used to perform full run-throughs of the configuration workflow. Complex algorithms are split into sub-algorithms. The presented artefacts of this section stem from an algorithm that consists of two sub-algorithms:

1. a vector normalization \(y[i] = a[i] \cdot x[i] + b[i]\) and
2. a mean squared error \(mse = \sum_{i=1}^{N} c[i] \cdot (y[i] - d)^2\).

4.2.3.1 Data Flow Models

Comparing the data flow models to the original Simulink models showed that all VBBs, connections, and parameters were extracted. **Listing 2** exemplarily presents an excerpt of an imported Simulink model and comprises a connection realized via the `<line>` tag, a VBB realized via the `<msblock>` of the type `MS_BB_AddSub`, its input and output ports, and a parameter `PARAM_OPERATION_ADDSUB` configuring the VBB to perform a subtraction instead of an addition. The presented parameter matches the earlier example of a code generation template in Listing 1.

4.2.3.2 Mapping Models

The VBBs of the imported data flow models were mapped onto the HWBBs of an IP-XACT model describing the quantities, the types, and the features of the HWBB as well as the interconnect of a DFA hardware accelerator. **Figure 11** shows an example for such a mapping: The mapping model is opened in the tree editor, which lists several mapping entries. The selected entry contains a reference to an IP-XACT component instance, which represents a load HWBB, and a reference to a load VBB. The entry defines also the execution unit and register set of the load HWBB via the properties `set` and `split`. If multiple `sets` are available, mapping data flow operations onto the same `split` but different `sets` results in the execution unit being shared.

4.2.3.3 Code generation

Finally, the code generator was provided with the IP-XACT-based hardware architecture models, the data flow models, and the mapping models. The resulting SystemC and VHDL test benches were successfully compiled and executed/simulated. With respect to the

**Listing 2** An excerpt of the imported Simulink model of an algorithm computing a vector normalization and the mean squared error.

**Figure 11** A mapping file opened in the editor, which was generated via the EMF.
excerpts of a template in Listing 1 and a data flow model in Listing 2. Listing 3 presents the generated configuration of a HWBB in the SystemC model: A HWBB is accessed via an array of pointers that uses the splits, sets, and registers as indices. The configuration value REG\_MASK\_OPERATION\_SEL\_ADDSUB\_SUB is written to the operation register BB\_EX\_ADDSUB\_REG\_ID\_\_OPERATION of the first set of the first split of the HWBB BB\_Ex\_fAddSub. Hence, the HWBB BB\_Ex\_fAddSub subtracts its incoming operands.

```plaintext
Listing 3 An excerpt of the generated SystemC code to configure a HWBB of a DFA hardware accelerator to perform a subtraction. The names of the constants were adapted to enable a proper layout.

```}{

\begin{verbatim}
1_BB_Ex_fAddSub
 [BB_SUBSPLIT_0][BB_SET_0]
 [regId + BB_EX_ADDSUB.REG_ID.OPERATION]
 =
 REG_MASK_OPERATION_SEL_ADDSUB_SUB;
\end{verbatim}

5 Conclusion and Discussion

The DFA is a CGRA that features runtime-reconfigurable and heterogeneous base blocks. These base blocks are connected via a sparse crossbar and offer complex configuration options. Besides, the DFA targets the next generation of ECUs. Manual configurations, a compiler for a high level language, such as C++, a specific ISA, and a model-based approach were considered to transfer algorithms onto the DFA. The application in the automotive field demands for the acceleration of selected algorithms and a high performance per size ratio instead of the support of arbitrary algorithms. Hence, the advantages of an interactive optimization and the high accessibility of a model-based approach outweighed the disadvantages, especially the required effort per algorithm.

This paper proposed a modeling framework based on Simulink and the DFA to model and simulate algorithms. The pivotal features of the modeling framework are:

- a library of VBBs and algorithms that offer different levels of abstraction allowing both, DFA and non-DFA experts, to model new algorithms and
- a simulation behavior that matches the behavior of the DFA on the functional level.

The modeling framework was validated by the successful modeling and simulation of several algorithms ranging from simple mathematical operations, such as matrix multiplications, to complex Bosch-internal algorithms with series relevance. This paper proposed also a configuration workflow to transfer a modeled algorithm onto actual DFA hardware. The configuration workflow relies on three types of models: data flow models, hardware models, and mapping models. Eclipse and EMF were used to generate Java code, including a simple editor, to load, store, and manipulate these models. As Xpand handles multiple model files per template well, Xpand templates and respective utility classes were implemented to generate the configuration and test benches in VHDL as well as SystemC. The configuration workflow was successfully validated based on the same range of algorithms as the modeling framework. These algorithms were imported into the data flow models and mapped onto the HWBBs. The data flow models, mapping models, and hardware models were passed onto the code generator. The resulting configurations and test benches in SystemC and VHDL were successfully compiled and executed.

All in all, the concepts of the modeling framework and the configuration workflow were successfully realized as prototypes. These prototypes proved the proposed model-based approach viable. Naturally, prototypes are by no means complete implementations: So far, the modeling framework supports only two hierarchical layers. Thus, an algorithm may not consist of sub-algorithms, but only of VBBs. Furthermore, the generated editors of the configuration workflow are simple tree editors. The Eclipse Graphical Modeling Project promises an easy development of graphical editors based on the EMF and the Graphical Editing Framework. Currently, the mapping must be performed manually. Automatizing the mapping would allow for a completely automatic configuration generation apart from the creation of the model of the algorithm via the modeling framework. These open issues and opportunities as well as a model-based optimization framework for designing different DFA hardware accelerator architectures are future work.

6 Literatur


