Chapter 10
Bridging Algorithm and ESL Design: MATLAB/Simulink Model Transformation and Validation

Liyuan Zhang, Michael Glaß, Nils Ballmann and Jürgen Teich

Abstract MATLAB/Simulink is today's de-facto standard for model-based design in domains such as control engineering and signal processing. Particular strengths of Simulink are rapid design and algorithm exploration. Moreover, commercial tools are available to generate embedded C or HDL code directly from a Simulink model. On the other hand, Simulink models are purely functional models and, hence, designers cannot seamlessly consider the architecture that a Simulink model is later implemented on. In particular, it is not possible to explore the different architectural alternatives and investigate the arising interactions and side-effects directly within Simulink. To benefit from MATLAB/Simulink's algorithm exploration capabilities and overcome the outlined drawbacks, this work introduces a model transformation framework that converts a Simulink model to an executable specification, written in an actor-oriented modeling language. This specification then serves as the input of a well-established Electronic System Level (ESL) design flow, enabling Design Space Exploration (DSE) and automatic code generation for both hardware and software. We also present a validation technique that considers the functional correctness by comparing the original Simulink model with the generated specification in a co-simulation environment. The co-simulation can also be used to evaluate different quality numbers of implementation candidates during DSE. As a case study, we present and investigate a torque vectoring application from an electric automotive vehicle.

Keywords Electronic System Level (ESL) · MATLAB/Simulink · Model transformation · Model validation · Design Space Exploration (DSE) · SystemC · Model-Based Design (MBD) · Code generation · SysteMoC · Torque vectoring

L. Zhang (✉) · M. Glaß · N. Ballmann · J. Teich
Hardware/Software Co-Design, Department of Computer Science,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: liyuan.zhang@cs.fau.de

M. Glaß
e-mail: glass@cs.fau.de

J. Teich
e-mail: teich@cs.fau.de

M.-M. Louërat and T. Maehne (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 311,
DOI 10.1007/978-3-319-06317-1_10

10.1 Introduction

Driven by the rapid development of microelectronics technology, the functionality and, thus, the design complexity of modern distributed embedded systems are continuously increasing. To cope with these challenges, *Electronic System Level* (ESL) [21] design methodologies introduce higher abstraction and model the complete embedded system as an *executable specification* at system level. At this level, the decisions such as the partition of functional units to software or hardware are yet to be made. Therefore, *Design Space Exploration* (DSE) [20] allows for an early evaluation of design decisions and searches for optimized implementation alternatives. After DSE, various tools, cf. to Gerstlauer et al. [11], are available to (semi-)automatically synthesize the implementation into software and hardware. Overall, ESL design helps the designer to deliver optimized systems and to shorten the design cycle.

In domains such as control engineering and signal processing, the development of an embedded system typically starts with the application engineer using domain-specific modeling tools such as *MATLAB/Simulink* [25] to build a functional model, e.g., a controller in a feedback control system. The application engineer often also uses Simulink to model the physical environment and to create the test bench for design validation. Simulink allows rapid design and is therefore often used to carry out algorithm optimization in early design stages. By using *Simulink Coder* [24], a Simulink model can be automatically translated into embedded C code for software implementation or HDL code for hardware implementation. However, there is no information about the architecture that a Simulink model is later implemented on. Therefore, considering different implementation alternatives and investigating architectural interferences and side-effects directly within Simulink is not possible.

In this work, we aim at closing the gap between classic ESL design flows and Simulink models by applying 1. *model transformation* and 2. a *system-level validation* technique (see Fig. 10.1). We employ an actor-oriented modeling language (*SysteMoC* [10]), which is based on *SystemC* [12], the de-facto standard for system-level modeling, to serve as the intermediate representation of the Simulink models and the input of an ESL design flow. Representing a Simulink model in an actor-oriented fashion is very suitable due to the nature of Simulink modeling, cf. to Lee and Neuendorffer [19]. Here, we introduce a model transformation framework (Sect. 10.4) that automatically generates an executable specification in SysteMoC from a given Simulink model. This executable specification is transformed with a component library to an exploration model. The exploration model is used within a Design Space Exploration (DSE) to consider different implementation candidates, which are evaluated by the DSE framework with respect to multiple design objectives and constraints. DSE delivers a set of high quality implementation candidates, from which the designer may subsequently choose the best trade-off as the system-level implementation for subsequent design phases. Moreover, we propose to validate the correctness of the automatic generated SysteMoC model using a co-simulation approach. We consider a control application from an electronic automotive vehicle

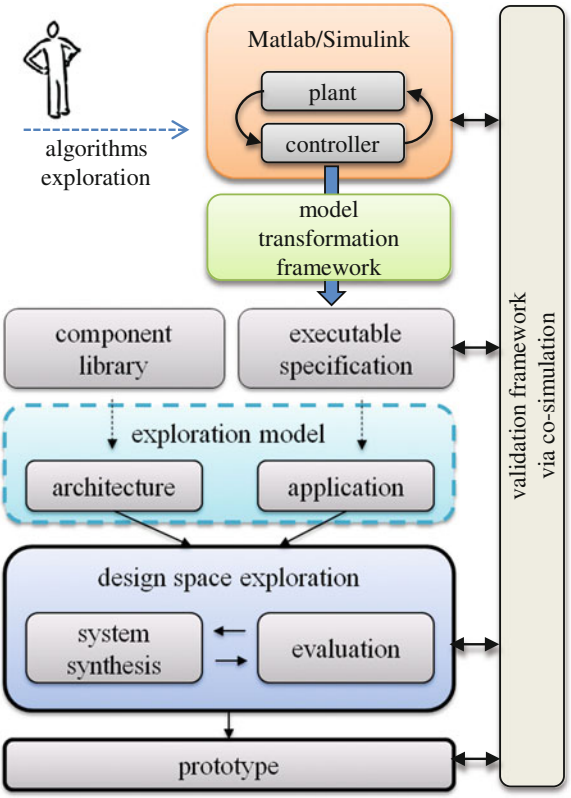

Fig. 10.1 Proposed design flow from MATLAB/Simulink to prototype via automatic model transformation and validation via co-simulation

to give evidence of the effectiveness of the proposed approach in Sect. 10.5 before we conclude the chapter in Sect. 10.6.

10.2 Related Work

Several contributions relevant to this work have been made in recent years by various research groups. Caspi et al. [5] focus their work on designing embedded software by translating Simulink models to *SCADE/Lustre* [8]. This intermediate representation is then implemented on the *Time Triggered Architecture* (TTA) introduced by Kopetz [17], which is a platform for running safety critical applications. Czerner and Zellmann [6] try to combine SystemC and MATLAB/Simulink for cycle-accurate hardware modeling and system verification by integrating SystemC modules into Simulink via *S-Functions* [22]. A co-simulation framework between SystemC and MATLAB/Simulink is built by Boland et al. [4] with the purpose of hardware

verification of DSP-based designs. There, Simulink is used to model the environment and to generate real-world stimuli to drive the *design under verification* that is implemented in SystemC. MathWorks also has a commercial tool (*HDL Verifier* [23]) to verify hardware designs using HDL simulators and FPGA *hardware-in-the-loop* test-benches. Above works only focus either on software design or on hardware design.

Kai et al. [14] present a Simulink-based MPSoC design flow by composing the Simulink model into a *Combined Algorithm and Architecture Model* (CAAM). The CAAM model can be then implemented at different abstraction levels using a multi-threaded code generator. Atat and Zergainoh [1] propose to refine a Simulink model at three different abstraction levels: transactional model, macroarchitecture model, and microarchitecture model. The system verification is carried out at all abstraction levels. These works try to enable system-level design with Simulink models. However, the partitioning of functional blocks into software or hardware must be performed manually by the designer.

Jersak et al. [13] introduce an approach to transform time-driven Simulink models into *System Property Intervals* (SPIs) to enable system-level timing analysis. They propose to transform time-driven models to data-driven models by combining register and virtual FIFO queues to guard the data exchange in multi-rate systems. Baleani et al. [2] use the *Synchronous Reactive* (SR) model as intermediate layer to enable the model transformation between MATLAB/Simulink, SR, and the model-based development tool set *ASCET* [9], which enables automatic code generation for automotive applications. In contrast to these works, we also consider domain- and application-specific knowledge during model transformation, where either a data-driven or a time-driven Model of Computation (MoC) can be chosen.

Another option to transform MATLAB/Simulink models is offered by employing *SystemC Analog/Mixed-Signal (AMS) extensions* [3], enabling the modeling of continuous systems within a SystemC-based design. In particular, it enables the transformation of *hybrid systems* (i.e., systems containing continuous and discrete state) from Simulink to SystemC. Since Simulink is no longer required to simulate the continuous part of the system, the simulation speed compared to the co-simulation technique increases. However, it may often not be desired or even possible to transform the huge and possibly closed-source continuous environmental models from Simulink to SystemC. Moreover, the result of such transformation does not come with the same benefits with respect to DSE.

In this work, we propose 1. a model transformation framework to automatically generate an executable specification from a Simulink model and 2. a system-level validation technique. Since our executable specification serves as the input for a well-established ESL tool flow introduced by Keinert et al. [15] that enables DSE, highly-optimized system implementations that satisfy given constraints can be achieved automatically. The advantage of our validation technique lies in re-using the test bench (including the environment model) created in Simulink.

10.3 Design Fundamentals

This section introduces the basics of Simulink and the actor-oriented model used to represent a Simulink model as an executable specification.

10.3.1 Simulink

Simulink is a commercial software from MathWorks [25] for modeling, simulation, and analysis of dynamic systems. Simulink is widely used to solve problems in automotive applications, communications, electronics, and signal processing. The basic elements in Simulink are *blocks* and *lines*. Basic blocks are mandatory units to perform computation or display functions, such as *Add, Memory, Scope*, etc. The designer builds hierarchical systems by encapsulating basic blocks into *subsystems*. Lines (also called *edges* or *channels*) are used to connect blocks and have *register* semantics (*non-destructive read, destructive write*). Besides these basic building elements, Simulink also has rich libraries that offer a broad variety of predefined blocks and can be used together with user-defined functions. Simulink provides several *solvers* to compute models that contains continuous and/or discrete states. It is very efficient to use Simulink during early design stages for algorithm exploration. By using *Simulink Coder*, a Simulink model can be automatically translated into highly-optimized C code for software implementation or HDL code for hardware implementation.

10.3.2 Executable Specification

In [10], a library for modeling and simulation of actor-oriented behavioral models termed *SysteMoC* is introduced. SysteMoC is based on SystemC, the de-facto standard for system-level modeling, adding actor-oriented MoCs to form executable specifications. In actor-oriented models, *actors*, which encapsulate the system functionality, are potentially executed concurrently and communicate over dedicated abstract *channels*. Thereby, actors produce and consume data (so-called *tokens*), which are transmitted by those channels.

- An actor (see the example in Fig. 10.2) is a tuple $a = (I, O, F, R)$, containing a set of actor ports partitioned into a set of actor input ports I (e.g., i_1) and a set of actor output ports O (e.g., o_1, o_2), a set of functions F (e.g., *fpositive, fnegative*), and a *Finite State Machine* (FSM) R. Actors can be grouped together to form *graphs*. A graph may also contain other graphs to build a hierarchical system.

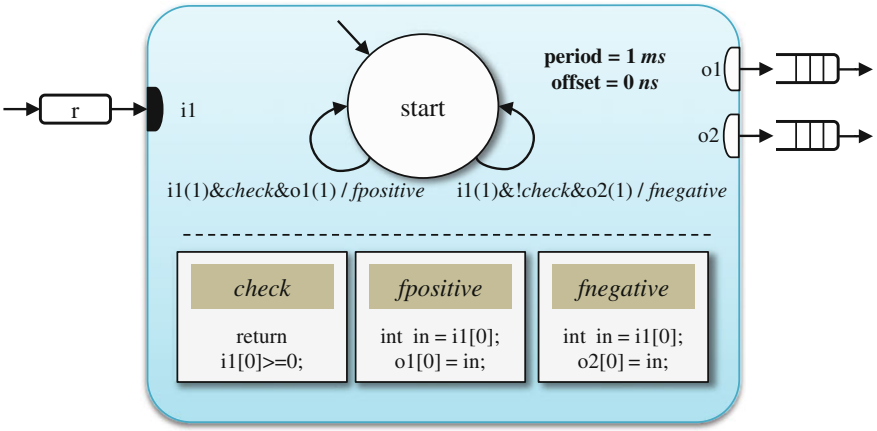

Fig. 10.2 A graphical representation of a SysteMoC actor, which sorts a sequence of input data depending on its algebraic sign

- The functions F encapsulated in an actor are partitioned into so-called *actions* f_{action} and *guards* k and are executed during a *transition* of the FSM R that also represents the communication behavior of the actor (i.e., the number of tokens consumed and produced for each transition).
- A transition is a tuple $t = (q_{\text{src}}, k, f_{\text{action}}, q_{\text{dst}})$ containing the source state q_{src} before the execution of the transition, and the destination state q_{dst} after the execution of the transition. An action f_{action} (e.g., *fpositive*) performs a computation task for the actor and may consume or produce tokens on the channel. A guard k (e.g., *check*) checks the availability of a transition by returning a Boolean value and the assignment of one or several guards to the FSM implements the required control flow. The firing of an actor corresponds to the execution of exactly one transition of the actor. If multiple actors have transitions that can be fired, they are chosen non-deterministically by the SysteMoC runtime system.
- A channel is a tuple $c = (I, O, n, d)$, containing a set of channel ports partitioned into a set of channel input ports I and a set of channel output ports O, its buffer size $n \in N_\infty = \{1, \ldots, \infty\}$, and a possibly empty sequence $d \in D^*$ of initial tokens, where D^* denotes the set of all possible finite sequences of tokens. In SysteMoC, actors are only permitted to communicate with each other via channels, to which the actors are connected by ports. Hence, in a SysteMoC actor, the communication behavior is completely separated from its functionality.

Figure 10.2 gives a graphical representation of an actor, which sorts a sequence of input data arriving on input port i_1 to either output port o_1 or o_2 depending on its algebraic sign. The actor reads its input data from a register r and writes its output data into two FIFOs. The actor has only a *start* state. Transitions of the finite state machine R are depicted as directed edges in the actor. Each transition is annotated with an activation pattern, a Boolean expression which decides if the

transition can be taken, and an action which is executed once the corresponding transition is taken. Using parameters *period* and *offset* indicates that this actor is time-triggered.

SysteMoC exploits the *event-driven scheduler* from SystemC to manage the firing sequence of the actors according to their FSMs. The basic SysteMoC implementation uses channels with FIFO semantics to provide a unidirectional point-to-point connection between an actor output port and an actor input port. Using FIFOs as communication channels makes SysteMoC suitable to model data-driven systems (e.g., signal processing applications). In other areas such as automotive systems, tasks are often executed periodically, i. e. they are time-triggered. In order to improve this kind of systems, SysteMoC supports *periodic actors* and *register* channels (see Fig. 10.2). A periodic actor has two additional parameters: *period* and *offset*, which describe the time that the SysteMoC scheduler evaluates the FSM of the actor. A register channel has non-destructive read, destructive write semantics. This extension enables a time-triggered MoC, hence, the modeling ability is greatly enhanced. For example, in a typical embedded control system, the sensors can be modeled using periodic actors and the rest of computation blocks can be data-driven.

We use *actor-based design* to construct the executable models (Fig. 10.3). The key advantage of actor-based design is that the interaction between actors follows some kind of communication pattern, called Model of Computation (MoC) [18]. A certain model of computation is given by a predefined type of communication behavior and a scheduling strategy for the actors. Separating actor computation and actor communication gives the designer the ability to refine the communication at different abstraction levels. Thus, the model is ideal for ESL design, since at the system level, the decision about the hardware software partitioning has yet to be made. Design

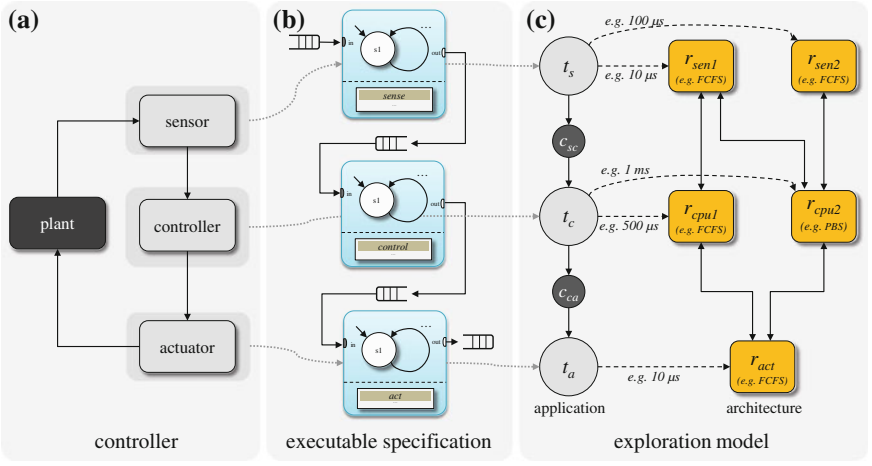

Fig. 10.3 To apply design space exploration for the controller (a), a graph-based exploration model (c) is automatically generated from the executable specification (b), a given architecture, as well as mapping constraints between them

Space Exploration (DSE) follows the commonly accepted Y-chart approach [16]. For each implementation, multi-objective optimization is used to evaluate the implementation quality. In the end, we obtain a set of high quality candidates as optimized implementation solutions.

10.4 Model Transformation

After the initial modeling in Simulink is finished, a block diagram of the system (e.g., containing controller and plant) is at hand. The Simulink model is verified via simulation. If the simulation results meet the design goals, the model transformation can be started.

10.4.1 Model Transformation Preparation

The first step of model transformation is *transformation preparation*, which changes the interface of a model that is going to be transformed (see Fig. 10.4). The part that is going to be transformed to SysteMoC (e.g., the controller) is connected with the environment model (e.g., the plant). Before model transformation, the designer must disconnect the chosen part from the environment (e.g., disconnect the controller from the plant) and then re-connect every input at the top level of the chosen part with an *Inport* block and every output at the top level with an *Outport* block. These I/O blocks form the interface of the chosen part. After model transformation, these I/O blocks are mapped to special actors that enable the data exchange between Simulink and SysteMoC.

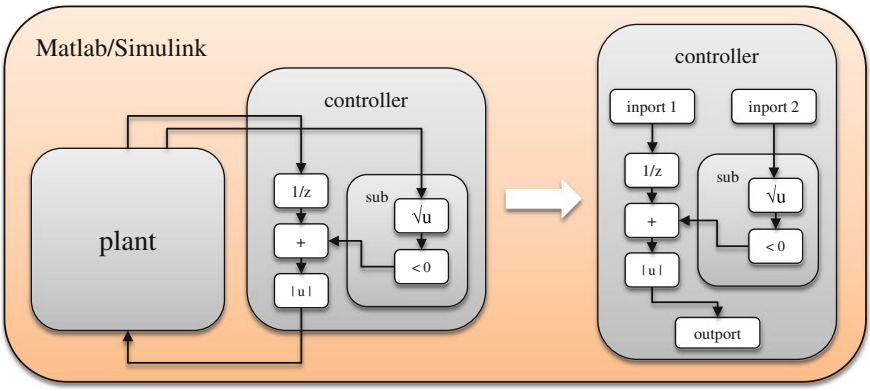

Fig. 10.4 The automatic model transformation requires a preparation step, in which those parts that shall be transformed (e.g., the controller) are disconnected from the other parts (e.g., the plant) and re-connected via I/O blocks

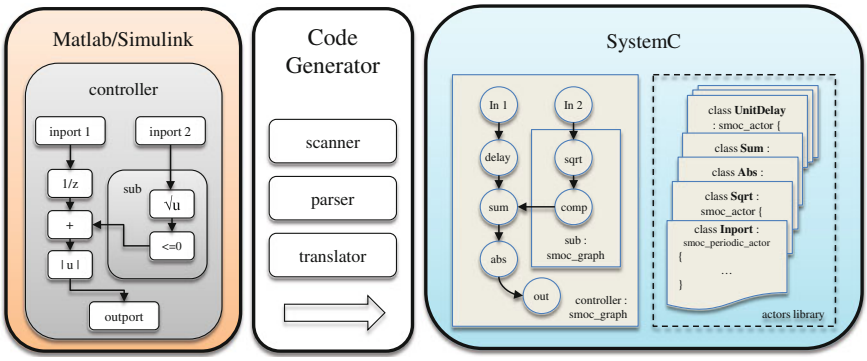

Fig. 10.5 The model transformation framework consists of an actor library and a Code Generator that maps Simulink basic blocks and lines to SysteMoC actors and FIFO channels

10.4.2 Model Transformation Framework

The core of the model transformation framework is a *Code Generator* that takes Simulink as input language and produces SysteMoC as output language. The structure of the Code Generator is shown in Fig. 10.5. The Code Generator has the most common operations in compiler design, such as lexical analysis, parsing, and code generation. There are three building parts:

- The *Scanner* reads the Simulink block diagram and filters the basic information elements.
- The *Parser* analyzes and identifies the semantics of the elements (e.g., basic blocks, lines). All the necessary information needed for model transformation is determined here (see Sect. 10.4.3), which includes the hierarchy and topology of the Simulink block diagram.
- The *Translator* determines the targeted MoC for the current Simulink block diagram (see Sect. 10.4.4). Additionally, the translator is responsible for the SysteMoC code generation.

10.4.3 Evaluating Simulink Block Diagrams

The Code Generator reads the source code of a Simulink block diagram in the form of an *mdl*-file and abstracts all the necessary information by using the *scanner* and the *parser*. The elements in an *mdl*-file currently supported by the Code Generator are *atomic blocks*, *subsystems*, *reference*, *user-defined blocks*, *lines*, and *branches*. We do not consider Simulink models with *algebraic loops*. An atomic block (i.e., Simulink basic block) represents a basic computation or display function. A subsystem contains multiple Simulink atomic blocks as well as subsystems. A reference is used to link an

atomic block or a subsystem. It contains, therefore, only a path to the implementation, which is normally stored in a library file (e.g., *simulink.mdl*). The Code Generator supports user-defined blocks, e.g., *S-Function* written in *C/C++*, by generating a wrapper in the SysteMoC model (as a placeholder). However, the designer has to integrate the implementation of the S-function into the wrapper.

A Simulink line represents a communication channel and has register semantics. Each line can have branches, which represent the multiple destinations. The multi-driver for a connection (i.e., one line, multiple source blocks) is forbidden in Simulink, but multicast of signals is allowed (i.e., one line, multiple target blocks).

All the necessary information described above are parsed by the Code Generator. The Code Generator also determines the *I/O data types* of each Simulink block. All signals by default have *double* types except 1. the types of the signals are implied from the block type or 2. the types are specified by the designer.

10.4.4 Model Transformation

The main task of the *translator* is to transform the Simulink model to SysteMoC with the targeted MoC. If the Simulink model is a *single-rate system*, all blocks share the same sampling rate. For this kind of models, it is straightforward to represent them as data-driven SysteMoC models. The Code Generator maps each Simulink block to a SysteMoC actor, whose function code is stored in the *actors library* (see Fig. 10.5). Each Simulink subsystem is mapped into a SysteMoC graph. Each Simulink line that has a point-to-point connection is mapped to a SysteMoC FIFO, which is a unidirectional point-to-point connection. For each Simulink line that enables multicast, a *multicast actor* is added between the source actor and the destination actors. This additional actor only serves as a relay for transmitting the data to all destination actors.

If the Simulink model is a *multi-rate* system, the blocks are sampled at different rates. For example in Fig. 10.6a, two Simulink blocks $b1$ and $b2$ are connected through a line l. Block $b1$ has the sampling rate λ and block $b2$ has the sampling rate λ/n. Block $b1$ and block $b2$ are transformed to actor $a1$ and actor $a2$, respectively. There are several options to represent this Simulink model in the SysteMoC environment:

10.4.4.1 Rate Transition Actors

Adding a *rate transition actor* c allows to explicitly coordinate the data exchange between $a1$ and $a2$ (see Fig. 10.6b). Two FIFOs are used to connect $a1$ and $a2$ with c. Per activation, $a1$ produces one token on the left FIFO, c reads one token from the left FIFO and produces n tokens on the right FIFO, $a2$ reads one token from the right FIFO. This solution is easy to implement but consumes additional memory space and causes extra delay.

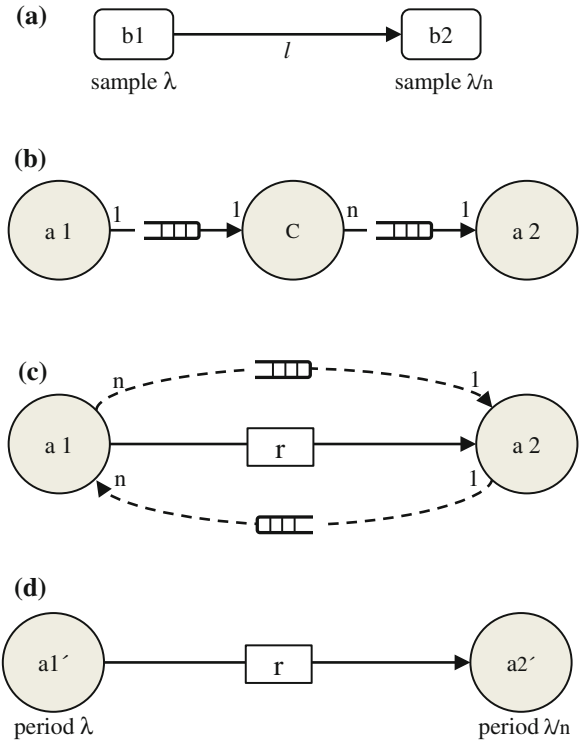

Fig. 10.6 A Simulink multi-rate model, as shown in (a), can be represented in SysteMoC by: adding a rate converter actor (b); transforming time-driven models to data-driven models (c); or creating time-driven SysteMoC models (d)

10.4.4.2 Data-Driven Transformation

Applying the technique introduced in [13] indicates adding a register channel r and two *virtual* FIFOs to govern the activation of $a1$ and $a2$ (see Fig. 10.6c). Activation of $a1$ and $a2$ is enabled by the availability of tokens on the two virtual FIFOs. This solution replaces the absolute periodic timing in Simulink models with relative execution rates. Thus, the time-driven Simulink MoC is transformed into a data-driven model. However, adding two virtual FIFOs per connection may increase the complexity of the design.

10.4.5 Time-Driven Transformation

Here, we propose to transform Simulink multi-rate models to time-driven SysteMoC models in order to preserve the simulation semantics of Simulink models. Blocks $b1$ and $b2$ are mapped to periodic actors $a1'$ and $a2'$. Line l is mapped to a SysteMoC

register channel r (see Fig. 10.6d). The sample rate of the Simulink block corresponds to the period of the periodic actor. Here, the activation of $a1$ and $a2$ is no longer governed by the availability of data on the communication channel. Because SysteMoC is originally designed to mainly model data-driven applications, the data dependencies (or topology dependencies in Simulink) are automatically preserved by the FSMs and FIFOs. But, this is no longer given when time-driven semantics are applied: If the two periodic actors have the same period, the SysteMoC runtime scheduler will not consider the partial order dependencies, hence, either actor that can be first simulated. Thus, the simulation behavior of SysteMoC may differ from Simulink. As a remedy, we propose to introduce artificial *offsets* for the periodic actors to reflect partial-order dependencies in time-driven SysteMoC. The Code Generator assigns a proper offset automatically for each periodic actor by running an analysis of the topology dependencies. In summary, the proposed model transformation technique can be divided into three parts:

1. using periodic actors to enable time-driven simulation,
2. using register channels to preserve the Simulink communication semantics,
3. using artificial timing offsets to include partial-order dependencies into SysteMoC models.

No matter which MoC is applied, each Inport block added during transformation preparation (Sect. 10.4.1) is mapped to a periodic actor. These periodic actors can be seen as hardware sensors (fetching the states of the environment model in Simulink). The sample rate t_a in a sensor actor is specified by the designer. Each Outport block added during transformation preparation is mapped to an actor, which is a representation of an actuator (sending the computation results back to Simulink). These sensor and actuator actors are typically grouped together to form a *co-simulation interface* for the auto-generated SysteMoC model.

10.5 Case Study: Torque Vectoring

In this section, an automotive application is used to evaluate the accuracy and efficiency of the proposed model transformation framework. Torque Vectoring (TV) is a new driver assistance system that distributes torque sent to each wheel to suit driving conditions and road surface in order to get more traction in curves. In this work, the Automotive Simulation Models (ASMs) of dSPACE [7] are used for modeling an electric rear-wheel drive vehicle and the environment in Simulink. A torque vectoring differential is realized by modifying the ASM to contain two basic engines (Fig. 10.7), so the left-side engine controls the torque sent to the left rear wheel, and the right-side engine for the right rear wheel, respectively. These two engines are controlled by a *torque vectoring controller*, which is implemented by an application engineer. A simple testing maneuver and the driving conditions are configured in ASM. The driving scenario is as follows: 1. the vehicle first remains motionless at position [0, 0] for 2 s; 2. the vehicle starts to accelerate and keeps a straight cruise without any steering;

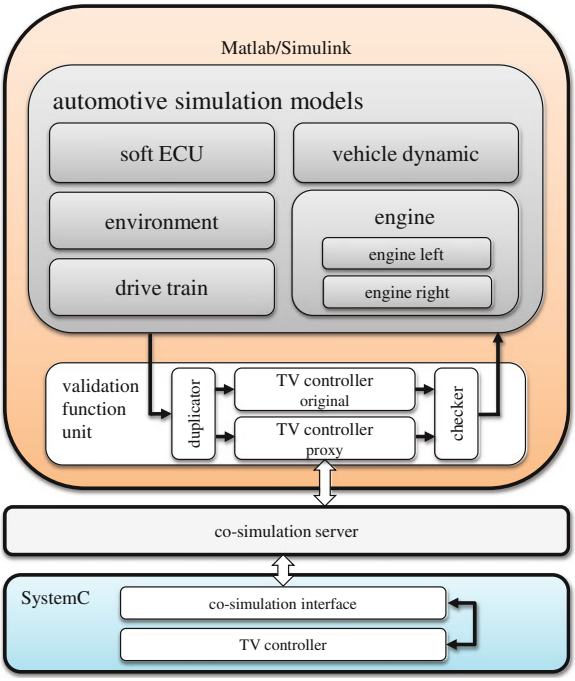

Fig. 10.7 Overview of the co-simulation between torque vectoring controller in SysteMoC and environment in MATLAB/Simulink for validation

3. the acceleration lasts for 20 s and the speed of the vehicle reaches 90 km/h at the new position [400, 0]; 4. the driver performs a *step steering*. This action lasts for 1 s and causes the steering wheel to turn left for 100°; 5. the steering wheel keeps its position for the rest of the scenario.

Simulating this maneuver in Simulink (with a continuous solver) first with TV enabled and then with TV disabled, the changes of the vehicle's position (Fig. 10.8) show that using TV controller shortens the radius while turning.

After the initial modeling and validation in Simulink, the TV controller is transformed to SysteMoC by the model transformation framework. Since TV is a single-rate system, the data-driven SysteMoC is used. The auto-generated torque vectoring controller contains 10 graphs, 98 actors, 110 FIFOs, and 770 lines of code (excluding library code).

After converting the Simulink model to an executable specification, a validation function unit is created in Simulink. Next, the co-simulation server and co-simulation interface are configured. The co-simulation checks whether the auto-generated TV controller will perform as intended in its operational environment. The testing maneuver configured in ASM is reused to evaluate the generated TV controller. The result of the co-simulation is given in Fig. 10.9: While the vehicle turns, in order to get more traction to shorten the curve, each engine is applied

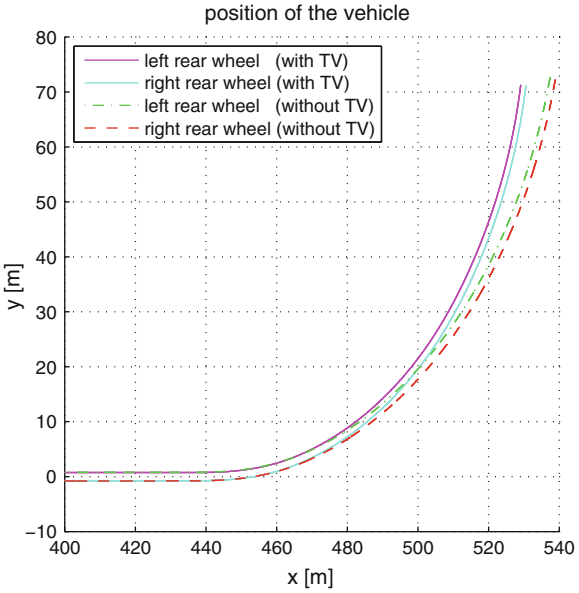

Fig. 10.8 Effect of torque vectoring on the vehicle's position: simulation results with TV controller enabled (*solid lines)* and disabled *(dashed lines)*

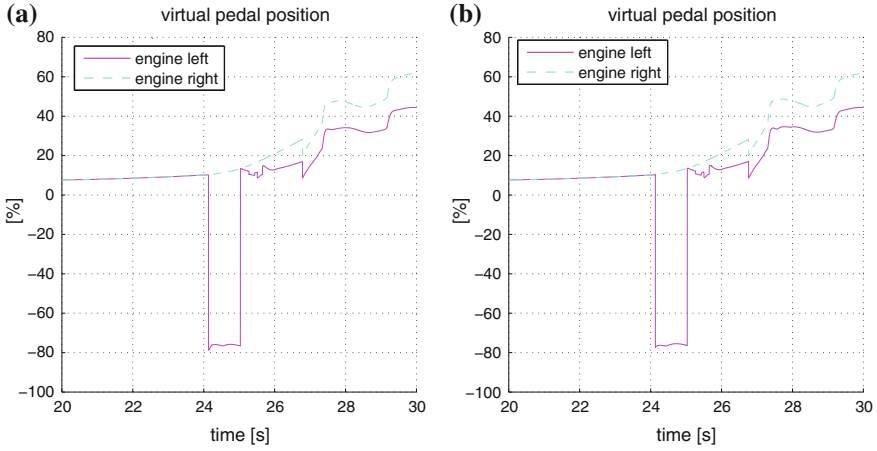

Fig. 10.9 Validation via co-simulation of Simulink and SysteMoC using the virtual pedal position seen by the engine: the proposed co-simulation shown in **(b)** delivers almost identical results compared to the plain Simulink simulation depicted in **(a)**

with an individual pedal signal calculated by the TV controller based on the current physical pedal position. These pedal signals are interpreted by the engines as *virtual pedal positions*, with a positive position for acceleration and a negative position for deceleration. Figure 10.9a shows how these two pedal signals change while using

the Simulink TV controller. When the vehicle is turning, the engine that generates torque for the inner wheel (in this case the left-side engine) actually receives a signal from the TV controller indicating a negative pedal position, this causes the inner wheel to brake while the outer wheel is still experiencing acceleration. As depicted in Fig. 10.9b, the SysteMoC TV controller delivers almost identical simulation results. The co-simulation shows a minor simulation deviation, which is observed by the validation function unit through comparing the pedal signals (see Fig. 10.9) calculated by the original TV controller and the generated TV controller. The small deviation shown in Fig. 10.10, which is always below 0.09 %, indicates a high accuracy of the proposed model transformation.

The development time for the presented case study is shown in Fig. 10.11: The development time consists of (I) implementing the initial model in MATLAB/Simulink (mandatory); (II) validating the initial model within the Simulink environment (mandatory); (III) automatic model transformation (proposed); and (IV) validation of the auto-generated model via co-simulation (proposed). The first two mandatory phases of modeling and validation in Simulink consume almost 70 % of the complete development time. The proposed automatic model transformation that converts the Simulink TV controller into SysteMoC requires 20 % of the overall development time—a high value at first glance. The reason is that the actors library did not yet include all the atomic blocks used in the Simulink TV model. Thus, the designer had to implement those Simulink atomic blocks and add them to the actors library. Note that this effort has to be invested only once for an atomic block. It is expected that the time consumption of this phase reduces dramatically

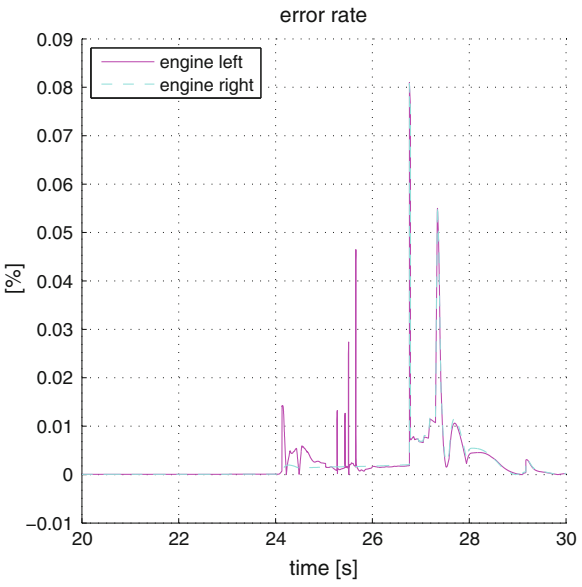

Fig. 10.10 The relative simulation error for the virtual pedal position remains below 0.09 %

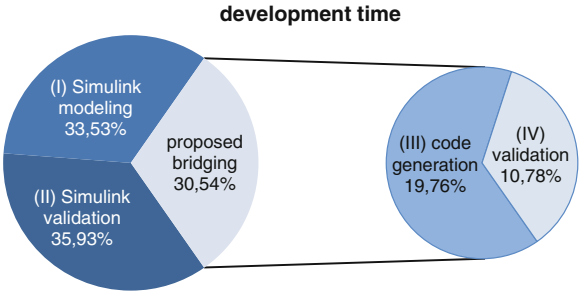

Fig. 10.11 The proposed methodology reduces the time from the Simulink model to the ESL model to only ≈30 % of the overall development time

for future applications (ideally to 0 %), given the constant extension of the actors library. But, this phase may include the integration of user-defined S-functions in the ESL model such that an application-dependent amount of development time remains to be taken into account. The last phase of system-level validation consumes about 10 % of the overall development time, resulting from the creation of the validation function unit and the configuration of the co-simulation framework. For the presented case study, it can be concluded that bridging algorithm design to automatic ESL design flows requires less than 50 % of extra development time compared to the mandatory MATLAB/Simulink part. Note that a huge amount of the extra development time arose from implementing basic blocks and configuring the co-simulation. Both aspects do not scale with the complexity of the modeled application, such that for future large and complex Simulink models, the extra effort for bridging to ESL design flows will become almost negligible.

10.6 Conclusion

In this chapter, we presented a framework that enables an automatic model transformation from MATLAB/Simulink to an actor-oriented design language (SysteMoC), which enables Design Space Exploration (DSE). This framework is integrated into an ESL design flow to further reduce development efforts. The automatic generation of an executable specification from Simulink has freed system designers from converting Simulink functional models into implementation models manually. On the other hand, by applying the proposed system validation technique, which is based on the co-simulation of the original Simulink model and SysteMoC model, the designer can easily validate the correctness of the executable specification. Furthermore, combining this work with DSE allows the designer to automatically get a first-hand evaluation on the performance of different implementation alternatives.

Acknowledgments The work has been partially supported by EFRE funding from the Bavarian Ministry of Economic Affairs (Bayerisches Staatsministerium für Wirtschaft, Infrastruktur, Verkehr und Technologie) as a part of the "ESI Application Center" project.

References

1. Atat Y, Zergainoh NE (2008) Automatic code generation for MPSOC platform starting from SIMULINK/MATLAB: new approach to bridge the gap between algorithm and architecture design. In: Proceedings on the international conference on information and communication technologies: from theory to applications (ICTTA) 2008, IEEE, pp 1–6
2. Baleani M, Ferrari A, Mangeruca L, Sangiovanni-Vincentelli AL, Freund U, Schlenker E, Wolff HJ (2005) Correct-by-construction transformations across design environments for model-based embedded software development. In: Proceedings of the design, automation and test in Europe conference (DATE) 2005, IEEE, pp 1044–1049
3. Barnasconi M, Einwich K, Grimm C, Maehne T, Vachoux A (2011) Advancing the SystemC analog/mixed-signal AMS extensions. Open SystemC Initiative (OSCI)
4. Boland JF, Thibeault C, Zilic Z (2005) Using MATLAB and Simulink in a SystemC verification environment. In: Proceedings of the design and verification conference (DVCon) 2005
5. Caspi P, Curic A, Maignan A, Sofronis C, Tripakis S, Niebert P (2003) From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. ACM Sigplan Not ACM 38:153–162
6. Czerner F, Zellmann J (2002) Modeling cycle-accurate hardware with Matlab/Simulink using SystemC. In: The European SystemC users group meeting (ESCUG) 2002
7. dSPACE (2013) Automotive simulation models (ASM). dSPACE. http://www.dspace.com/
8. ESTEREL (2013) SCADE suite™: control and logic application development. ESTEREL technologies. http://www.esterel-technologies.com/
9. ETAS (2013) ASCET. ETAS. http://www.etas.com/
10. Falk J, Haubelt C, Teich J (2006) Efficient representation and simulation of model-based designs in SystemC. In: Proceedings of the forum on specification and design languages (FDL) 2006, pp 129–134
11. Gerstlauer A, Haubelt C, Pimentel AD, Stefanov TP, Gajski DD, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circ Syst 28(10):1517–1530
12. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publisher, Boston
13. Jersak M, Cai Y, Ziegenbein D, Ernst R (2000) A transformational approach to constraint relaxation of a time-driven simulation model. In: Proceedings of the international symposium on system synthesis (ISSS) 2000, IEEE Computer Society, Washington DC, pp 137–142. doi:10.1145/501790.501820
14. Kai H, Sang-il H, Popovici K, Brisolara L, Guerin X, Li L, Yan X, Chae SI, Carro L, Jerraya A (2007) Simulink-based MPSoC design flow: case study of motion-JPEG and H.264. In: Proceedings of the ACM/IEEE design automation conference (DAC) 2007, pp 39–42
15. Keinert J, Streubühr M, Schlichter T, Falk J, Gladigau J, Haubelt C, Teich J, Meredith M (2009) SYSTEMCODESIGNER—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Trans Des Autom Electron Syst 14(1):1–23
16. Kienhuis B, Deprettere E, Vissers K, van der Wolf P (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors (ASAP) 1997, pp 338–349. doi:10.1109/ASAP.1997.606839

17. Kopetz H (2011) Real-time systems: design principles for distributed embedded applications. Springer, Hermann
18. Lee EA (2000) What's ahead for embedded software? Computer 33(9):18–26
19. Lee EA, Neuendorffer S (2004) Actor-oriented models for codesign. Formal methods and models for system design. Kluwer Academic Publishers, Norwell, pp 33–56
20. Lukasiewycz M, Streubühr M, Glaß M, Haubelt C, Teich J (2009) Combined system synthesis and communication architecture exploration for MPSoCs. In: Proceedings of the design, automation and test in Europe Conference (DATE) 2009. IEEE Computer Society, Nice, pp 472–477
21. Martin G, Bailey B, Piziali A (2010) ESL design and verification, a prescription for electronic system level methodology. Morgan Kaufmann, San Francisco
22. MathWorks T (2013) S-Function. http://www.mathworks.com/
23. The MathWorks (2013) HDL Verifier™: Verify HDL and Verilog using HDL simulators and FPGA-in-the-loop test. The MathWorks. http://www.mathworks.com/
24. The Mathworks (2013) Simulink Coder: Generate C and C++ code from Simulink and Stateflow models. The MathWorks
25. The Mathworks (2013) Simulink: Simulation and model-based design. The MathWorks