Integration of Simulink Models with Component-based Software Models

Nicolae MARIAN, Søren TOP
Mads Clausen Institute for Product Innovation, University of Southern Denmark
Alsion 2, DK-6400 Sønderborg
nicolae@mci.sdu.dk

***Abstract*—Model based development aims to facilitate the development of embedded control systems by emphasizing the separation of the design level from the implementation level. Model based design involves the use of multiple models that represent different views of a system, having different semantics of abstract system descriptions. Usually, in mechatronics systems, design proceeds by iterating model construction, model analysis, and model transformation. Constructing a MATLAB/Simulink® model, a plant and controller behavior is simulated using graphical blocks to represent mathematical and logical constructs and process flow, then software code is generated. A Simulink model is a representation of the design or implementation of a physical system that satisfies a set of requirements. A software component-based system aims to organize system architecture and behaviour as a means of computation, communication and constraints, using computational blocks and aggregates for both discrete and continuous behaviour, different interconnection and execution disciplines for event-based and time-based controllers, and so on, to encompass the demands to more functionality, at even lower prices, and with opposite constraints. COMDES (Component-based Design of Software for Distributed Embedded Systems) is such a component-based system framework developed by the software engineering group of Mads Clausen Institute for Product Innovation (MCI), University of Southern Denmark.**

Once specified, the software model has to be analyzed. One way of doing that is to integrate in wrapper files the model back into Simulink S-functions, and use its extensive simulation features, thus allowing an early exploration of the possible design choices over multiple disciplines.

The paper describes a safe translation of a restricted set of MATLAB/Simulink blocks to COMDES software components, both for continuous and discrete behaviour, and the transformation of the software system into the S-functions. The general aim of this work is the improvement of multi-disciplinary development of embedded systems with the focus on the relation between control engineering and software engineering.

***Index Terms*—component-based design, model-based design, MATLAB/Simulink, model transformation, discrete-time models, continuous-time models**

I. INTRODUCTION

SIMULINK is a design-based modeling tool, even a framework, created by MathWorks, that uses graphical blocks to represent mathematical and logical constructs and process flow [2], for the modelling, simulation and analysis of dynamic systems. To the control system engineers the structure and content of the system representation in Simulink is intrinsic and intuitive and the design can be layered depending on the analysis level required.

Simulink is a block diagram industry oriented standard tool for simulating mixed reactive/transformative, nonlinear dynamic systems that builds on the MATLAB environment for technical computing. A Simulink model is a representation of the design or implementation of a system that satisfies a set of requirements. For many years Simulink has been the tool of choice for much of the control industry, by many considered a de-facto standard, to develop both physical and control system models, in terms of stability, response time, overshoot, etc [2,6], and is maturing into the new generation of systems engineering, representative of an advanced approach to design. The main attraction of Simulink has been its flexibility and the range of toolboxes available to aid control system design, development and calibration.

COMDES (as a family of versions) is a component-based framework intended for design and analysis of embedded control systems [1]. Specific for COMDES is the design of the control system from prefabricated components, and then analyzing the executable models that constitute its configuration specification. COMDES offers strong typing, explicit initialization, explicit time management (delays, clocks, etc), and simple expression of concurrency (data dependencies), based on a well-defined control and data-flow representation. By means of a graphical signal flow graph editor, it supports model-based development. The end result is an integrated process of software development, featuring model-based configuration and analysis of embedded applications that can be characterized by the sentence:

What you specify is what you verify, execute and test.

COMDES (as in its last version COMDES-II) uses an actor diagram that represents subsystems (actors) and the signals exchanged between them within the corresponding distributed transaction. An actor encapsulates state and exhibits behaviour. Individual actor behaviour is described by means of reactive behaviour, which is usually specified with a state-machine model, and continuous behaviour, i.e. some numerical (conditioned) processing, that is specified as a function block diagram. The atomic software unit is a function block. Function blocks are reusable system components, which eventually enable the reconfiguration of system structure. Ultimately, the system actor diagram is transformed into a function block design for control and data.

Although the different disciplines are tightly coupled in the considered embedded systems, their development is often a rather sequential, mono-disciplinary, process. Typically, first the control part is designed, next the

Digital Object Identifier 10.4316/AECE.2008.02001

hardware infrastructure is fixed, and finally the embedded software is developed. This approach can create large problems, especially for the software engineers. For instance, choices about the placements of sensors (and implicitly the occurrence of interrupts), control rates, control delays, hardware, etc., have a strong influence on the complexity of the software. Moreover, usually many implicit assumptions are made, which first become visible at system integration. This easily leads to non-optimal solutions.

Within each discipline, a common solution is the frequent use of models to detect problems as early as possible. Using the model-based engineering paradigm for the design of a system, the models are preferably specified using domain-specific modelling formalisms. For instance, in the software domain a lot of effort is put on model driven development, based on component models. Moreover, mono-disciplinary modeling is usually supported by tools that allow some form of execution or simulation. Lacking, however, is the possibility to combine tools of different disciplines, to investigate the mutual influence of modeling choices. Our aim is to couple currently used tools to allow both automatic translation of the Simulink models to COMDES component-based models, but also simulation of the transformed software models into the Simulink setup [10,11]. By means of co-simulation, the component models can be combined to obtain a better overall system behavior.

II. MODEL-BASED DESIGN IN CONTROL AND SOFTWARE

Model-based design provides a proven technique for creating embedded control systems. Model-based system development is a change of focus from arithmetic and data related issues to the overall architecture of software system. It is a more effective approach to increase system functionality and reliability, and to decrease development cost and time.

The design of control algorithms is a fundamental part of the design flow. It starts from a functional specification and ends up with a detailed description of the algorithms. In the model-based design methodology, the part of the control algorithm that is mapped to the software partition is automatically translated from a model representation to a set of software components. The software architecture of the application will accommodate and compose together those software components such that the real-time requirements are met. In the proposed design flow, the control algorithms are captured using the MATLAB/Simulink design environment and the automatic translation of the model to COMDES-language code is performed according to a set of rules, described in a following section.

A *Simulink* block-diagram represents a dynamic system described as a set of first-order ordinary differential equations.

$$\begin{cases} dx/dt = f(x, u) \\ y = g(x, u),\ x(t_0) = x_0 \end{cases}$$

The system can then be specified as in the form above, where the vector u gives the input signals to the system, y gives the vector of output signals from the system and the vector x gives the state of the system.

A Simulink model is represented graphically by means of a number of interconnected blocks. Lines between blocks connect block outputs to block inputs and represent data flow signals. Blocks may have states, which may consist of a discrete-time and a continuous-time part.

The output of a block is computed by an output function, based on its input and its current state and time. Similarly, an update function calculates the next discrete state. A derivative function relates the derivatives of the continuous part of the state to time and the current values of the inputs and the state.

Blocks can be built from a large number of predefined library blocks, which can be nested in an arbitrary structure container composition, or they can be implemented by an S-function, which can be written in MATLAB, C, C++, Ada, or Fortran, belonging to Simulink's callback architecture. An S-function can be used for a variety of applications such as describing a system as a set of mathematical equations, incorporating existing code into a simulation, and adding a block that represent athe scheduler of a real-time kernel.

During the simulation of a Simulink model, the outputs, inputs and states are computed at certain intervals, from a start time to an end time, as specified by the user. The successive states of a system are computed by a so-called solver, a Simulink-specific program.

Since no solver is suitable for all models, there are several types of solvers. The solvers use numerical integration to compute the continuous states of a system from the state derivatives specified by the model. Each solver uses a different integration method, allowing the selection of the most suitable method for a particular model.

The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called step size. The step size depends on the type of the solver used, the characteristics of the Simulink model, and the existence of discontinuities of the continuous states (Simulink checks for such discontinuities – this is called zero crossing detection – and if it detects one within the current step, the precise time at which zero crossing occurs is determined and additional time steps are taken).

There are several types of solvers. Fixed-step solvers use a fixed step size. Variable-step solvers change the step size during simulation. They reduce the step size to increase accuracy when states are changing rapidly and increasing the step size to avoid taking unnecessary steps when states are changing slowly. This requires some additional computation each step, to determine the step size, but can reduce the total number of steps and hence the duration of the simulation.

For purely discrete models there are discrete solvers. Continuous solvers compute continuous states using numerical integration. Simulink provides an extensive set of fixed-step and variable-step continuous solvers, each implementing a specific numerical integration technique for solving the ordinary differential equations that represent the continuous states of dynamic systems. The solvers monitor the error at each time step; they compute the local error, which is the estimated error of the computed state values. If the local error is greater than the acceptable error for any state, the solver reduces the step size and tries again.

Simulation of a Simulink model starts with the initialization phase, where e.g. library blocks are incorporated, block parameters are evaluated, memory is allocated and the execution order of the blocks is determined. Next, Simulink enters a simulation loop, consisting of simulation steps. During each simulation step, Simulink executes all blocks of the model in the order determined during initialization. This execution order does not change during the simulation. For each block, Simulink calls functions that compute the block's states, derivatives, outputs for the current sample time, and the next time step. This continues until the simulation is complete.

COMDES (*Com*ponent based design of software for *D*istributed *E*mbedded *S*ystem) which is being developed at MCI, University of Southern Denmark, is an instance of component based design, which is a software design method for real-time embedded systems. COMDES is an executable visual model for embedded control systems. It mirrors the architecture of the control process itself, which is made up of a number of independent control modules, called function blocks, with a top-level module controller, wired by signals, such as pressure, temperature, etc. The COMDES system design is meant to be a diagram of components together with a description of their interconnection topology and/or state based behavior. The native timing execution of the model is a periodically clocked events pattern (under a more general paradigm called timed multitasking), [1].

III. TRANSLATION OF SIMULINK MODELS TO COMDES MODELS

Our approach to translate bottom-up and hierarchical the controller part of the Simulink design, covering functional and timing aspects, is based on:

- mapping tables between a safe set of Simulink blocks and the corresponding COMDES function block model.
- a set of translation rules formulated for both Stateflow and Simulink conditional blocks for converting of some implicitly behaviour into an explicitly one:
  - derive COMDES state machine model out of the Stateflow state machine, associated input and output variables and conditional blocks like *Switch*, *Multiport* switch, and
  - associate the continuous part of Simulink to the states of the COMDES state machine model in terms of the function blocks (basic and composite, respectively).
- mapping of types and timing constraints
- translation of the Simulink design into the COMDES software design using an intermediate *XML*-based model, which represents a COMDES model in terms of class, attributes, and node-based hierarchies, according to the rules established in the previous steps.

The semantic of Simulink is multiple, as presented in the previous section, dependent on user-defined options, like that of the simulation method. Simulink has a problem with the typing (weak), and lacks modularity sometimes. COMDES design and implementation has to filter out Simulink ambiguities, and to serve as a reliable middle layer for safety critical applications.

We start our translation with a Simulink model, consisting of two parts: a discrete-time part describing the controller and a continuous-time (or perhaps discrete-time) part containing the environment in which the controller is supposed to operate. Modeling both the controller and the environment is of course essential for studying the properties of the controller by simulation. Once the design is validated by various simulation methods, the implementation of the controller can start. Figure 1 shows a glance of how they look like, i.e. Simulink and COMDES models. One of the actors of the COMDES model is called *Controller*, and is namely the result of the translation from the Simulink model. The software model is having other actors like *Sensor*, for reading the inputs, *Actuator*, for actuating the outputs, and *Operator Station*, for man-machine interface and communication, which are not functionally derived but timing constrained in the translation.

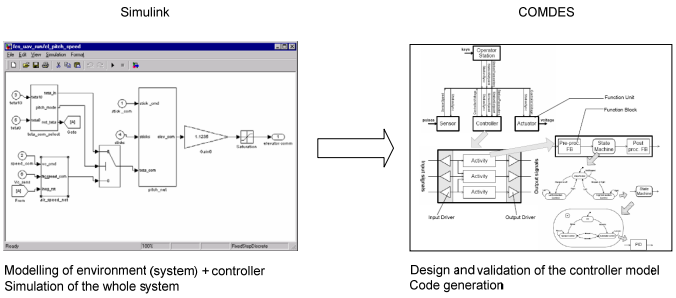

Figure 1. Simulink and COMDES models.

Both Simulink and COMDES allow the representation of signals and systems, more precisely, multi-periodic sampled systems. The two languages share strong similarities, such as a data-flow model and similar abstraction mechanisms (basic and composite components). However, there are several differences:

1. COMDES has discrete-time semantics, whereas Simulink has continuous-time semantics. It is important to note that even the *discrete-time library* of Simulink blocks produce piece-wise constant continuous-time signals. In a continuous-time model, signals continuously vary with time and the blocks respond to continuously changing input. In a discrete-time model, signals are sampled at discrete time intervals; input and output take place in cycles.
2. COMDES has a unique, precise semantics. The semantics of Simulink depends on the choice of a simulation method. For instance, some models are accepted if one chooses variable-step simulation and rejected if one chooses *fixed-step*, *auto*, or *multithreaded* simulation.
3. COMDES is a strongly-typed system with explicit type set on each flow. In Simulink, explicit types are not mandatory. A type-checking mechanism exists in *Simulink* (some models are rejected due to type errors) but, as with the execution semantics, it can be modified by the user by setting some *flags*.
4. COMDES is modular in certain aspects, whereas Simulink is not: for instance, a *Simulink* model may contain implicit inputs (the sampling periods of a system and its sub-systems, which are not always inherited).

Given the above differences, the goals and limitations of our translation when mapping a Simulink block-diagram into COMDES elements are the following:

1. We only translate a discrete-time, non-ambiguous part of Simulink. In particular, we do not translate blocks of the continuous-time library, *S-functions*, or MATLAB functions. Each Simulink block is mapped into one COMDES block, according to built mapping tables. The Simulink model to be translated is assumed to be (part of) the controller embedded in a larger model (including the environment).
2. The translation is restricted to the following simulation method of the solver: *fixed-step*, *discrete* and mode: *auto*. We assume that the Simulink model to be translated has the *boolean logic signals* flag *ON*. Then, a requirement on the translator is to perform exactly the same type inference as Simulink. In particular, every model that is accepted by Simulink must also be accepted by the translator and vice versa.
3. The COMDES program must be run at the time period the Simulink model was simulated. Thus, an outcome of the translation must be the period at which the COMDES program shall be run (i.e., the period of the basic clock). To know the period at which the Simulink model was simulated, we assume that for every external input of the model to be translated the sampling time is explicitly specified. Simulink concept of periodic execution is maintained for time-continuous blocks, as they can be mapped to COMDES in the same way as blocks with a discrete sample time.

For reasons of traceability, the translation must preserve the hierarchy of the Simulink model as much as possible. The task of finding a good granularity is also important for an efficient translation. Simulink ports are mapped into *Sensor*, *Actuator*, and *Operator Station* actors of the COMDES model. Simulink edges are mapped into COMDES signals, so that to maintain Simulink destructive write, nondestructive read, and broadcast semantics. Simulink model of computation is thus transformed into a state oriented data-driven model, with partial ordering between blocks, which is supported by COMDES. In this respect, we follow, in general, the ideas from [4,5,7,12], for graph transformation in terms of states and dataflow, instantiated to component-based diagrams embedded into states, and intend to use in the future the tool GReAT [8] to automate the process.

Mapping tables

Each box in a *Simulink* diagram is called a block; the wires carry signals. The inputs and outputs of a system are represented by rounded boxes containing numbers. Typically, a block takes some input signals and produces some outputs according to a function determined by the kind of block in question. There are libraries of blocks, and they can also be user-defined. The rectangular boxes without inputs output the constant value they display. The circles are sum blocks. Boxes enclosing names are subsystems; they denote control systems defined in other diagrams. Blocks can have state. For example, blocks labelled 1/z are unit delay blocks. They store the value of the input signal, and output the value stored in the previous cycle. In each cycle, the output depends on the values of the inputs and of the state that may be held in the blocks, but other factors may be relevant. For example, subsystems may be conditionally executed: an action subsystem has an activated input and is executed when it is true; an enabled subsystem has an enabling input and is executed when its value is greater than zero. When a subsystem is not executed, its outputs can either be held at their previous value or reset to an initial value. Any state contained in blocks within the subsystem is held until the subsystem is about to be executed again, at which point the states can be held or reset to an initial value. Merge blocks take a number of inputs and produce one output: the most recently calculated input.

Translation of *Simulink* blocks (e.g., adders, multipliers, the 1/z transfer function) into basic *COMDES* function blocks is made in a bottom-up fashion, i.e., starting from the basic blocks. More complex *Simulink* blocks (e.g., discrete filters) are translated into *COMDES* composite function blocks.

From the whole *Simulink* block library we choose only a safe set of them: *Continuous*, *Discontinuous*, *Discrete*, *Signal Routing* and *Ports & Subsystems*.

Simulink		COMDES		
Block Name	Time Behaviour	FB Name	Functional Descriptions	Time Behaviour
Derivative	Continuous	Derivative	$y(k)=\frac{1}{\Delta t}(u(k)-u(k-1))$ Output equals the change in input value divided by the change in time.	Δt is the change in time since the previous simulation time step.
Integrator	Continuous	Integrator	$y(k)=\Delta t(u(k)+u(k-1))$ Output equals the sum of input value multiplied by the change in time.	Δt is the change in time since the previous simulation time step.
Transport/ Variable Transport Delay	Continuous	Transport Delay	$y(t)=u(t-t\_delay)$ delays the input by a specified amount of time. It can be used to simulate a time delay.	t is the current time,t_delay is the delayed time.

Figure 2. Mapping tables for the safe set of *Simulink* blocks.

The relationships between the *Simulink* blocks and corresponding *COMDES* function blocks are given in terms of a set of mapping tables, and an example is shown in the Figure 2.

If we take the continuous time integrator block, then it is one of a number of blocks that can function in a variety of different ways depending on the choices made by the user each time the block is added to a model. The options for the integrator block include applying limits to the output, initializing with internal or external initial conditions, allowing for external reset signals, outputting state information and information on the limit condition. To define all this in *COMDES* in a way that achieves the ideas, the software block contains the definitions for the input and output connections to have the required number of connectors for this method. For example, if an external initial condition is required then two inputs are needed rather than one.

This same structure idea has also been used for many other blocks including the discrete integrator, math function block, trigonometric function block and many others.

A basic function block specifies a simple subroutine function or a fragment of a complex one. It is ready to execute whenever the necessary signals arrive at its inputs.

Then the basic function block will execute, and produce signals at its outputs, which can be the input signals of the successor basic function block and so on, until the last one is executed and the final output signal is generated.

Two or more basic function blocks that aggregate a complex function can be seen as a composite function block. For example, we can specify a composite function block called *Substract_Max_Gain*, which is composed of basic function blocks such as *subtract*, *max*, *constant*, and *gain*, etc.

A composite function block is externally indistinguishable from a basic function block. It can be viewed as an aggregation of basic function blocks that execute in certain precedence by a standard routine called *function block driver*.

The *function block driver* is a static function block execution scheduler that linearly invokes encapsulated basic function block instances according to the execution schedule. The execution schedule is derived for each state from the signal flow diagram depending on *Simulink* application and translation rules.

Translation rules

The following aspects of the *Simulink* model of computation have to be preserved by translation:

1. causality and the resulting partial ordering of *Simulink* blocks,
2. relative execution rates between *Simulink* blocks, and
3. read and write access sequence on each edge to model the register semantics used by *Simulink* for communication.

The execution precedence of function blocks mentioned above is a control flow model (computation sequence graph), which is actually a state transition model, i.e. a Moore machine whose states are associated with the execution of certain function blocks. In this way, states specify different steps of the whole computation sequence, which is specified by transitions among different states and their relevant predicates/guards. Those transitions integrate a computation sequence path within the state machine.

Therefore the computation sequence graph can be implemented by means of a binary decision diagram table, and a function block execution sequence driver that interprets the table in a sequential logic controller like manner.

State Machine Model

A transition is the mapping that indicates the relationship between a source state and the target state, representing reaction or response. It is used for determining the execution sequence of computation.

A *Simulink* model can include a *Stateflow* block, which is defined by a diagram that has local data and includes finite state machines, flow-diagram notations, and state-transition diagrams. The finite state machine reacts to events triggered in the *Simulink* model; the reactions lead to state changes that affect the behavior of the *Simulink* model.

A simpler situation occurs when there is only a *Switch* block in the control part of a *Simulink* diagram, Figure 3, or a *Multiport* switch block, with similar translation.

In *COMDES*, the transitional relationship is expressed in a structure called next-state mapping, which clearly shows all the information included in a transition described above.

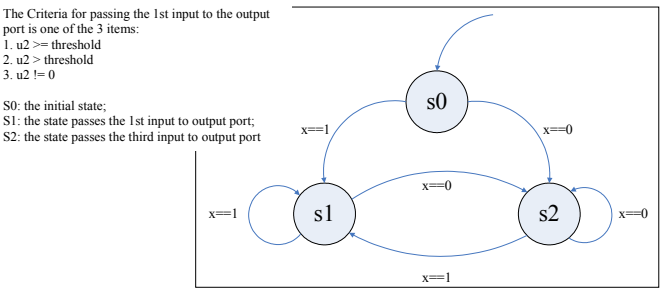

Figure 3. Translation of the *Switch* block.

Translation rules for the *Stateflow* diagram are similar with those in [7], namely identification of the states and transitions in a new state machine, using the current state machine of the *Stateflow* diagram, input variables from different blocks, and output variables wired with a number of *Switch* blocks, called switching signals. A simplification using binary vectors and associated operators is acomplished.

The translation algorithm according to the formulated rules is presented below:

1. Each state s_i is split into A = $2^{notdefined(s_i)}$ locations, where *notdefined*(s_i) is the number of switching signals *not* defined in s_i. The total number of the switching signals is D. The set of locations generated from s_i is
$$\Sigma_i = \{\sigma_{i,1}, \sigma_{i,2}, \ldots \sigma_{i,D}\}.$$
2. Make a transition $\tau_{i,n,j,m}$ between $\sigma_{i,n}$ and $\sigma_{j,m}$ if
$$(C_{i,n} \oplus C_{j,m}) \wedge M(\sigma_{j,m}) = <0>$$
where $C_{i,n}$ is the binary switch code for location $\sigma_{i,n}$, $C_{j,m}$ is the binary switch code for location $\sigma_{j,m}$, and $M(\sigma_{j,m})$ is the mask code for location $\sigma_{j,m}$.
3. Choose $\sigma_{1,1}$ to be the initial location.
4. Add the switching signal values from the entry action of s_i as the following invariants to location $\sigma_{i,j}$.
5. Delete all unreachable locations and transitions from the new state machine.
6. For each state s_i generate function blocks (composite). Copy these function blocks (composite) from each state s_i to corresponding locations $\sigma_{i,j}$, for all j = 1, 2, 3, ... , D, in the new system.

Type & Timing inference

Typing inference is that from *Simulink* model to a C language, i.e. Boolean, integer, real. Whenever two types are compatible but not the same, the stronger type will be modified accordingly.

As related to time, a large proportion of the blocks in the *Simulink* standard library can run in different time-modes, i.e. either continuous or discrete time modes. In addition where blocks are able to run in discrete time mode they can be defined to run at a predefined sample rate or they can inherit their sample time from their parent system or from their driving block.

The translation relaxes the over-constrained exact timing assumed for all *Simulink* components, since

1. it is fundamentally impossible to implement blocks or channels with zero execution time and
2. activation of all blocks at exact points in time is unnecessarily restrictive.

thereby yielding a larger design space.

Timing constraints are (re)inserted afterwards to ensure time critical correct real-time implementation, typically I/O activities which have to satisfy external requirements. Additionally, latency path constraints, typically between inputs and outputs or along a cycle, may have to be specified to guarantee timely completion. The execution period of *COMDES* model will be the same with *Simulink* simulation timing (triggers). In periodically time triggered state machine, switch control action takes place in the state and in event-driven state machine it takes place in the entry action of the state.

IV. INTEGRATION OF THE COMDES BLOCKS INTO S-FUNCTIONS

In order for the simulation to capture accurate behavior of the subsystem software in real time, it is essential that the same software to be modeled in the simulation setup as well. Simulink uses block diagrams to model dynamic systems and provides an environment that allows simulation of COMDES component-based models by converting of software components into custom S-functions. Simulink provides the necessary templates to create an S-function, and the software models will be executed within the Simulink shell as S-functions, Figure 4.

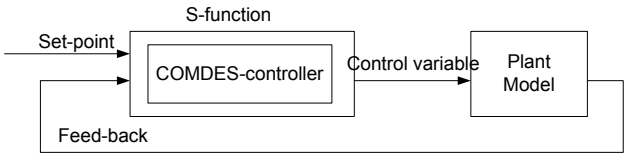

Figure 4. Encapsulation of a COMDES controller into a Simulink S-function.

Control engineers can simulate and thus verify the COMDES component-based model implementation by comparing its performance to that of pure SIMULINK models of the controller, and this can happen iteratively, allowing an early evaluation of the development process.

An S-function is a system function that has been compiled to run in the Simulink environment. On a Microsoft Windows® platform, these system functions are compiled into dynamically linked libraries (DLLs). The software component is then compiled and built using the MATLAB mex® function library. This build process within the MATLAB application produces a DLL for the software component. The MATLAB-produced DLL is then used in the simulation model by associating it with a user defined S-function.

This will allow to accurately simulate several modes of operation and to improve the control algorithms in the real application, once the simulation results met their targets.

An S-function implement a set of methods, called callback methods that together describe the behavior of the (sub) system modelled by means of the S-function. The simulation coordinator, in our case Matlab Simulink, invokes these callback methods during the simulation.

An implementation of the S-function interface must implement the set of callback methods. Some callback methods are optional.

The simulation coordinator invokes an optional callback only if the S-function defines the callback, by a C-code included at the end of the S-function. The callback methods perform tasks required at each simulation stage. These tasks performed by the callback methods include:

- Initialization. Prior to the first simulation loop, the simulation coordinator initializes the S-function. During this stage:
  - The SimStruct is initialized. A SimStruct is a simulation S global structure that contains information about the S-function. Some library functions permit to create in it variables usable to pass values among subsequent *mdlOutput* calls or to store state.
  - The number and dimensions of input and output ports are set.
  - The block sample times are set.
  - The storage areas are allocated.
- Calculation of next sample hit. In case a variable sample time block is specified, this stage calculates the time of the next sample hit; that is, it calculates the next step size.
- Calculation of outputs in the major time step. After this call is complete, all the output ports of the blocks are valid for the current time step.
- Update of discrete states in the major time step. In this call, all blocks should perform once-per-timestep activities such as updating discrete states for next time around the simulation loop.
- Integration. This applies to models with continuous states and/or non-sampled zero crossings. If the S-function contains continuous states, the simulation coordinator calls the output and derivative portions of the S-function at minor time steps.

In this way, the simulation coordinator can compute the states for the S-function. If the S-function contains non-sampled zero crossings, the simulation coordinator calls the output and zero-crossings portions of the S-function at minor time steps so that the zero crossings can be located.

The main callback function together with a brief description of their functionality is given in Table 1.

TABLE I. FUNCTIONALITY OF THE CALLBACKS METHODS

S-function	Description
mdlInitializeSizes	Specifies the number of inputs, outputs, states, and parameters and other characteristics of the S-function
mdlInitializeSampleTimes	Specifies the sample rates
mdlInitializeConditions	Initializes the state variables
mdlOutputs	Computes values of the output variables
mdlUpdate	Updates the state variables
mdlDerivatives	Computes the derivatives
mdlZeroCrossings	Updates zero-crossing vector
mdlTerminate	Performs any actions required at termination of the simulation

During simulation, Simulink invokes the predefined set of callback functions of the S-function implementation in order to perform all necessary computations in the right order.

Figure 5 shows the order in which the simulation coordinator invokes the callback methods of an S-function, during initialization and simulation respectively. Solid rectangles indicate callbacks that always occur during model

initialization and/or at every time step. Dotted rectangles indicate callbacks that may occur during initialization and/or at some or all time steps during the simulation loop.

A standard S-function contains a number of functions that are called by Simulink. For instance, during a simulation step, Simulink calls *mdlUpdate*() to update discrete states, *mdlDerivatives*() to calculate derivatives, and *mdlOutputs*() to calculate the outputs of a block. The execution phase of each Simulink block is an iterative computation of (1) the block outputs (2) block states and (3) the next time step. The function *mdlOutputs*() calculates the output of the block, while *mdlUpdate*() updates the block states.

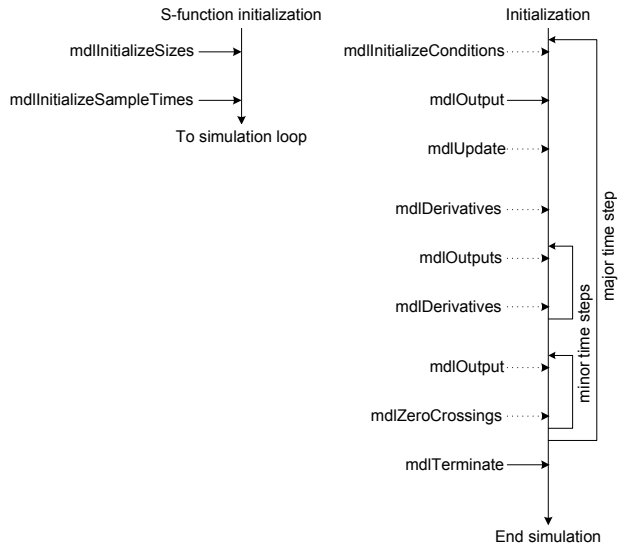

Figure 5. Initialization and simulation steps of S-function.

Code errors will be notified and serious problems caused by run-time errors are resolved; S-function code is executed by Simulink and errors often cause a Simulink failure, which often is solved only by a MATLAB reboot.

The Simulink model is becoming a plane of interconnected Simulink components, COMDES components, each with a representation in the Simulink plane, and their interaction, Figure 6.

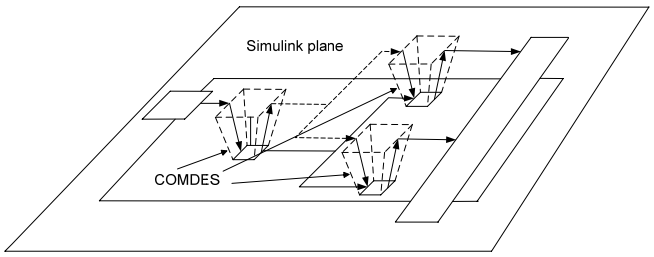

Figure 6. Addition of a COMDES plane in Simulink.

V. EXAMPLE OF SIMULINK – COMDES TRANSLATION AND CO-SIMULATION

In order to illustrate how to apply the translations rules defined in the former chapter, the tank level control example is taken [7], Figure 7.

Applying the transformation rules, we derive the *COMDES* state machine model first, and then the function blocks associated with states. The flat Stateflow state machine contains three states and three switching signals connected to the control input port of switch blocks *Switch1*, *Switch2* and *Switch3*. By applying the translation algorithm steps 1-4, we get a state machine with 8 states, and, finally after step 5, we end up with a 6 states state machine. The final step 6 attaches dataflow to the states.

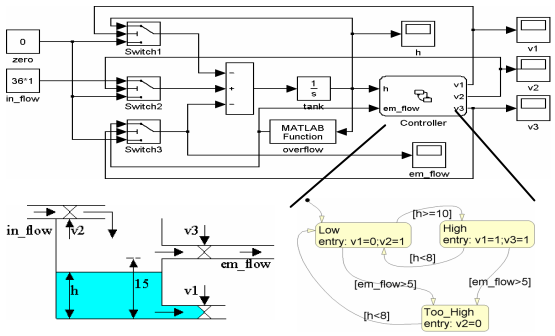

Figure 7. Tank level *Simulink* model.

For example, state *Low* has the following Simulink diagram, Figure 8.

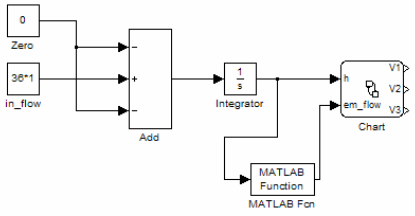

Figure 8. Simulink diagram for state *Low*.

We translate the MATLAB function into a Simulink diagram again, Figure 9.

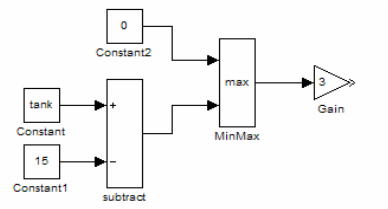

Figure 9. Simulink diagram for MATLAB function block.

Combining the last two diagrams, we have the resulting Simulink diagram for state *Low*, Figure 10.

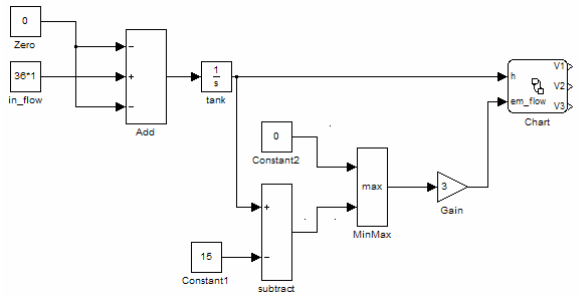

Figure 10. Final *Simulink* diagram for state *Low*.

We derive the COMDES function block diagram for state *Low* by applying the mapping tables and causality, as well as for the other states, following similar procedures, Fig. 11.

The simulation model is including the software model by wrapper file template, which in turn is compiled into a Windows DLL component and used by Simulink in performing simulations. Wrappers abstract COMDES details, but make them available at conceptual level.

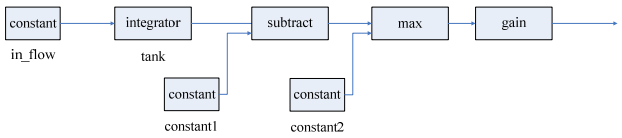

Figure 11. COMDES function block diagram for state *Low*.

The implementation of the callback methods of the COMDES S-function block are defined for each of the functions mentioned in the previous section. As an example, in the case of *mdlInitializeSizes*, the discrete variables of the process are mapped to the discrete state variables of the S-function block. The continuous variables and the variable time of the process are mapped to the continuous state variables of the S-function block. The number of input ports is set to the number of input variables that are specified in the formal parameter list of the model. The number of output ports is set to the number of output variables as specified in the parameters of the COMDES S-function block. The number of sample times is set to 1. The number of zero crossings is set to the number of occurrences of guard operators plus the number of occurrences of inequality delay predicates plus 1 for the time events.

mdlInitializeSampleTimes is implemented by setting the sample time of the S-function is set to a continuous sample time with offset 0. *mdlInitializeConditions* is implemented such that the initial values for the state variables from the S-function are obtained from the valuation from the process, using the variable mapping as defined in function *mdlInitializeSizes*. *mdlOutputs* is copying the values of the discrete and continuous variables and the variable time to the corresponding output variables. If the simulation step is a major time-step, it is determined whether a time event occurred. If the *mdlUpdate* function is called for the first time (*ssIsFirstInitCond*(S) holds, where S denotes the SimStruct), the process is simulated using the Simulink simulator. If a time event occurred (*IsTimeEvent* holds), which is determined in function *mdlOutputs*, the resulting process is obtained using the end-valuation of the time transition (*EndState*(*TimeStep*)). After that, this process is simulated using the simulator.

In the same way, other functions like *mdlZeroCrossings*, *mdlTerminate*, are implemented.

VI. CONCLUSION

In this paper we proposed a methodology of transforming Simulink control models into COMDES software component-based models. Model-based design, used to its fullest, provides translation-connected design environments that enable developers to use Simulink as a single model of their entire system for visualization and validation, and ultimately software product deployment, with consistent automatic code generation.

In this respect, some problem issues may be solved by a joint control engineer and software engineer effort. Even if the control and software design work is done in isolation, there is a close cooperation between the two expertises. The control engineers can concentrate on their specific issues, like damping, overshoots, stability of control, whereas the software engineers can focus on embedded aspects like scheduling issues, resource consumption, priorities. The interaction is directly between the two models, like the relation between the control loop execution periods and the task deadlines.

The advantages of using COMDES as a software design and implementation method of Simulink control models are the straightforward of the translation, well structured design, feasibility of verification and validation by combining different techniques, reusability of parts of these in relation to reuse of components, and cost-efficient development of code. We intend to improve the translation to handle other Simulink semantics.

The model-based development process benefits from control-software co-design, resulting faster and shorter development cycles and earlier testing. Maintenance of the control algorithms becomes easy because control algorithm models are safely translated into component-based software architectures, and wrapping back software components into a library of simulation components, Simulink is used as a testing tool too. In this way, a simulation tool independent representation of the control algorithms has been achieved. The new opportunities for checking and the clear division between control algorithm and simulation code has been appreciated.

REFERENCES

[1] N. Marian, “Model-Based Development of Embedded Software Systems with Components”, Advances in Electrical and Computer Engineering Journal, vol 1/2006, pp 30-38

[2] Simulink, A tool for modeling, simulation and implementation of control systems, see:
URL http://www.mathworks.com/products/simulink.

[3] M. M. Adams and P. B. Clayton. Clawz: Cost-effective formal verification for control systems. In 7th International Conference on Formal Engineering Methods, pages 465–479, 2005.

[4] P. Caspi et al., “Translating Discrete-Time Simulink to Lustre”, in *Proc. of the 3rd International Embedded Software Conference EMSOFT’03*, LNCS 2855, 2003, pp. 1-15

[5] N. Scaife et al., “Defining and Translating a “Safe” Subset of Simulink/Stateflow into Lustre”, report No TR-2004-16, Verimag, 2004

[6] T. Henzinger, C. M. Kirsch and M. A. A. Sanvido, “From Control Models to Real-Time Code Using Giotto”, IEEE Control Systems Magazine, Feb. 2003, pp. 50-64

[7] A. Agrawl, G. Simon, G. Karsai, “Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Translations”, in *Electronic Notes in Theoretical Computer Science*, vol 109, 2004, pp 43-56

[8] M. Andries, et al., “Graph Transformation for Specification and Programming”, Sci. Comput. Program., Vol. 34, No. 1, 1999, pp. 1-5

[9] IEEE Transactions LaTeX and Microsoft Word Style Files, Available: http://www.ieee.org/organizations/pubs/transactions/ stylesheets.htm

[10] S. Top, H.J. Nørgaard, B. Krogsgaard, B.N. Jørgensen, “The Sandwich Code File Structure - An architectural support for software engineering in simulation based development of embedded control applications”, Proceedings of IASTED International Conference on Software Engineering (SE 2004). ACTA Press, (2004)

[11] S. Top, H.J. Nørgaard, B.N. Jørgensen, “Object oriented C++ programming in SIMULINK® - A reengineered simulation architecture for the control algorithm code view”, Proceedings of Nordic MATLAB Conference 2003, pp 79-84

[12] I. Stürmer, D. Travkin,, “Automated Transformation of MATLAB Simulink and Stateflow Models”, Proceedings of 4th Workshop on Object-oriented Modeling of Embedded Real-time Systems, 2007, pp 57-62