Error-free Control Programs by means of Graphical Program Design, Simulation-based Verification and Automatic Code Generation

Stephan Seidel   Ulrich Donath
Fraunhofer Institute for Integrated Circuits
Design Automation Division
Zeunerstrasse 38
01069 Dresden, Germany

Stephan.Seidel@eas.iis.fraunhofer.de   Ulrich.Donath@eas.iis.fraunhofer.de

Abstract

Currently the formalisation in the process of creating automation control programs starts with the programming of the real-time controller. But inconsistencies in the requirements definition and misinterpretations will lead to errors in the program which have to be resolved through expensive software-in-the-loop and field tests. This paper introduces a holistic approach for the formalisation of the control design already in the design phase. It also illustrates the design flow for the model-based creation of error-free control programs. Created by means of graphical editors the system definition, which includes the control algorithm, is transferred into Modelica code and thus the executable system model is used for the simulation-based verification. The simulation results are compared to the requirements. Once these are fulfilled and no further errors found, program code for the real-time controller is generated automatically. In this paper Structured Text for programmable logic controllers (PLCs) according to IEC 61131 is generated. In final software-in-the-loop tests the real-time capabilities of the control program are validated.

Keywords: Graphical program design; Modelica; IEC 61131; Structured Text; Software in the Loop

1 Introduction

During traditional control program development it is often necessary to find erroneous sections in the software and fix such code modules by means of extensive software-in-the-loop simulations. The errors are often caused by wrong interactions between components of the control program which are a result of inconsistencies in the initial project definition [1]. Such inconsistencies could have been found by an intensive and paper-dominated reviewing process which is often shortened in order to save time. As an alternative the formalisation should no longer be carried out at the coding stage but already at the design stage of the software project.

The formalisation of the design consists of the creation of models which contain not only the structure of the system but also the definition of the functionality. In case the models are executable in a simulation software the simulation-based testing of single components as well as the overall system is feasible. Simulation results will be compared to the project's requirements definition. As soon as differences occur they have to be solved by an iterative adjustment of the models or a more precise process definition. When the overall models functionality complies with all requests from the definition an automatic code generation is executed during which code for the target controller is generated. For the final verification of the generated code software-in-the-loop tests are carried out. In these simulation-based tests the real control program is coupled with machine and operator model components which provide request and response signals.

The control program development which is described in the following sections is based on a case filling machine. This machine's model can be subdivided into the machine model, the control model, and the operator model [2].

The machine model is composed of models of all the relevant subsystems such as electrical, mechanical, and hydraulic systems. All of these models are coded in Modelica [3] and are available as objects in libraries. They are instantiated in the model view and then connected with each other in order to reproduce the structure and composition in the real world. The machine model acts in its entirety as a model of the physical environment with which the controls interact.

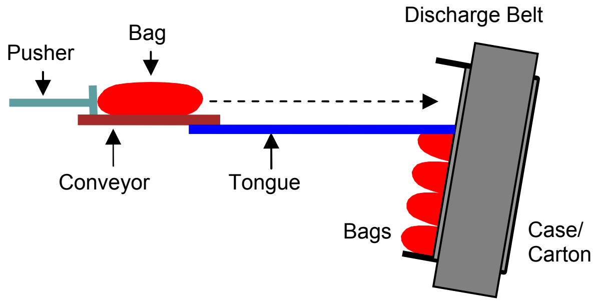

Figure 1: Schematic of the case filling machine

The control model describes the required control operations for the desired course of machine actions. In simple cases the control operations may be linear assignment sequences, but in complex cases they are algorithms which represent state machines. For the graphical description of state machines an UML-profile [4][5] is utilized. These statecharts include simple states, sequential composite states, and pseudostates (initial state, junction, shallow history). The transitions of states are triggered by signal triggers, change triggers, and time triggers and can also be labelled with guards and priority numbers. Priority numbers of transitions are an addition to UML in order to define an evaluation sequence. Activities, which are coded in Modelica, can be assigned to entries, exits, and transitions of states. They are also edited with the graphical editor while creating the statechart. The whole statechart will be automatically translated into Modelica code.

The operator model contains the operating instructions of the system which are derived from the production schedule. They form in general a command sequence and are also given as Modelica code in order to enable an overall system simulation with SimulationX [6].

Each step during the development of the control program

- Graphical program design
- Test and verification at Modelica level
- IEC 61131 code generation [7]
- Software-in-the-loop tests

is illustrated by using the example of the feeding device of the case filling machine.

This machine conducts the final packaging of bags containing products into shipping cartons. The bags are transported on a conveyor from the production process to the feeding device shown in Figure 1.

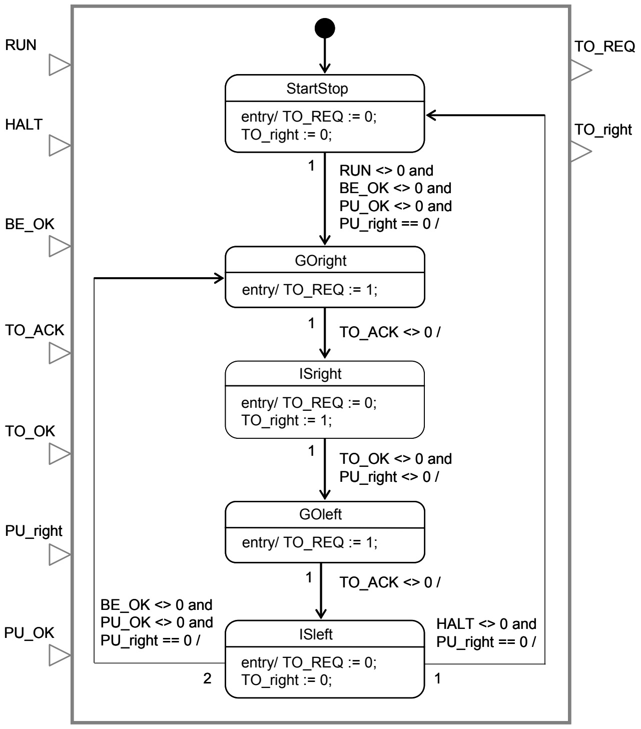

Figure 3: Statechart of the tongue controller

They are placed by the conveyor in front of a pusher on a tongue. The carton is positioned by a belt directly in front of the pusher. As soon as the carton has arrived at this position the tongue is extended and

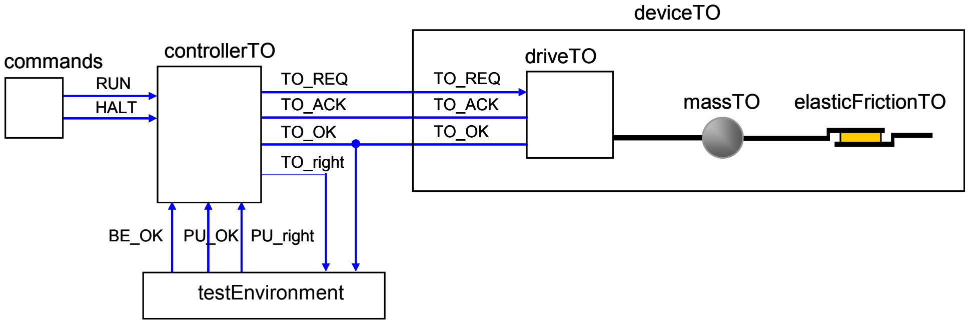

Figure 2: Structure of the tongue's model

the bag is pushed by the pusher over the tongue into the carton. Thereafter the tongue and pusher are retracted and the carton is repositioned to the next filling position.

This paper is organised as follows. Section 2 introduces the graphical design approach. Thereafter section 3 demonstrates the simulation-based verification of the system. The code generation for Modelica and IEC 61131 is detailed in section 4 whereas section 5 illustrates the software-in-the-loop simulation and its benefits. The paper closes with a conclusion in section 6.

2 Graphical program design

At first the models for *tongue (TO)*, *pusher (PU)* and *discharge belt (BE)* are designed. Each component is separated into its control model and machine model.

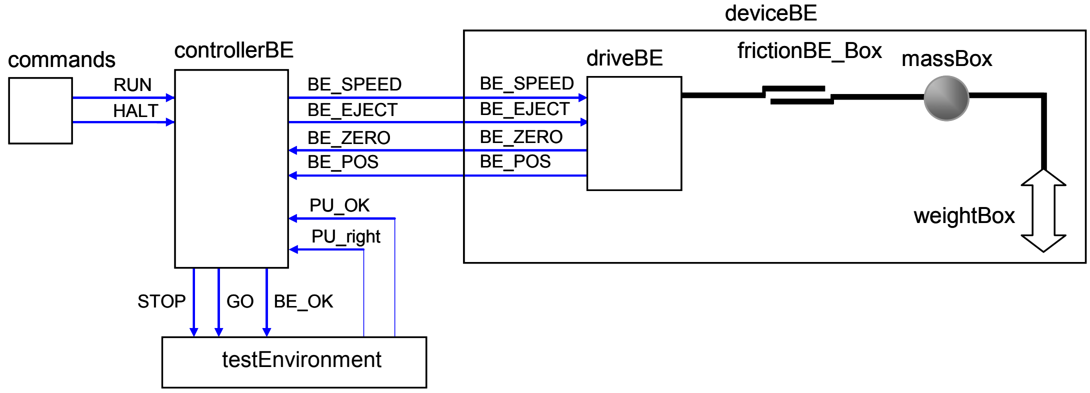

Figure 4: Structure of the discharge belt's model

The machine models of tongue and pusher have the same internal structure consisting of drive mechanism, mass, and friction and show an equal device interface (Figure 2). The setting of the component's input signal REQ (request) will start the positioning of pusher or tongue while the direction of the movement is alternated. The component's output signal ACK (acknowledgement) affirms the movement request whereas OK indicates the request's completion.

The interface of the control is modelled inversely to the device interface. Additional inputs are RUN and HALT to start and stop the operations (Figure 2). Furthermore signals are defined which contain the status of the cooperating devices or send the own status to other devices.

The controls of tongue and pusher are implemented as state machines (Figure 3). In the entry activities of the states the signal REQ is set or reset. The drive will move thereupon the tongue or pusher back and forth. Transitions between states contain Boolean conditions which act as change triggers. Such triggers fire when the corresponding condition is evaluated to true. In this example these triggers are the feedback from the drive ACK and logical combinations of OK, RUN, and HALT with status signals from the cooperating devices.

The model of the component discharge belt describes the up and down movement of belt and carton. Apart from drive, mass, and friction, the weight is also registered which accumulates as the carton is filled with bags. The interface (Figure 4) includes the signals SPEED, EJECT, ZERO, and POS. Input signal SPEED is the nominal value for the carton's velocity. After the carton is filled, the setting of input signal EJECT starts the ejection of the full carton and thereafter the insertion of an empty one. Output value ZERO indicates the arrival of the carton at the zero position whereas POS indicates its current position. The component's interface is completed with signals RUN and HALT and diverse status signals.

As shown before the control operation is described by a statechart (Figure 5) which groups the belt's movement into a starting phase (states: RunUp, SlowToMin, SlowToNull) and a cyclic positioning phase (RunMax, RunMin, SlowToNull) for filling the carton. In the entry activities of the states the belt's velocity SPEED is set and the target position is calculated. Transition triggers are the ZERO signal as well as the current belt position POS in relation to the target position. Right before the start of the movement the pusher's activity is checked to avoid damage caused by a collision between pusher and carton. The cyclic repositioning of the belt is stopped as soon as the carton is full and the carton is ejected.

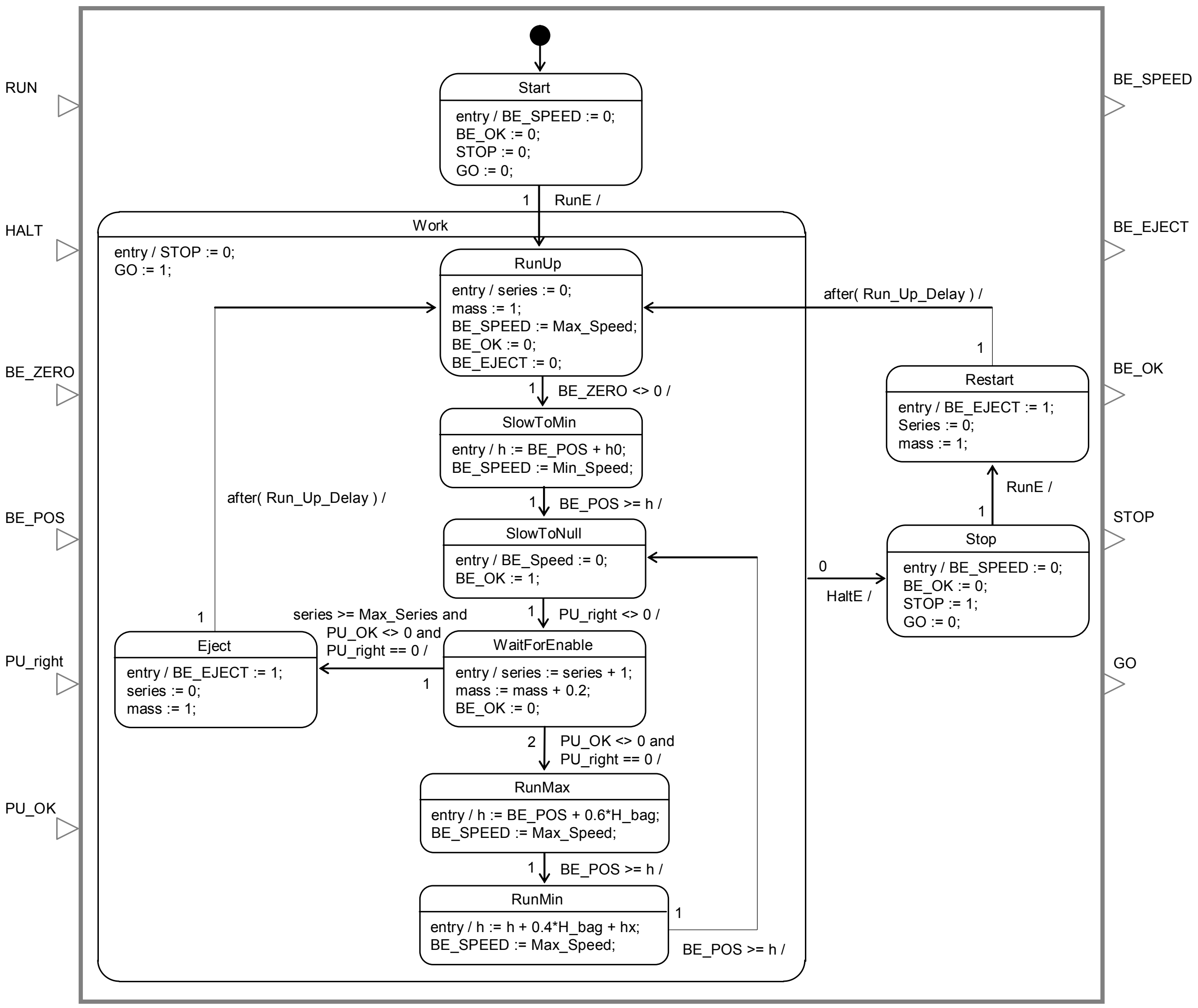

Figure 5: Statechart of the discharge belt controller

After the design of each component (pusher, tongue and discharge belt) they are instantiated in the model view and connected with each other (Figure 6). The belt controller takes over the master function by activating the GO and STOP signals as well as the OK signal after successful positioning for the subordinate controllers of pusher and tongue.

3 Test and verification at Modelica level

Models of the devices are built from physical elements such as mass and friction and from signal blocks which are also part of the library and possess a representation in Modelica. The control components are each translated automatically into a Modelica *model* with corresponding type, variable, parameter, and signal declarations as well as an *algorithm* section. The structure of this code is illustrated in section 4.1. Figure 6 shows the overall Modelica model which consists of all aforementioned components and their connections.

At first for the verification of the controls, the simulation results of pusher and tongue are compared to the requirements. In Figure 2 the structure of the tongue's model is shown. The model of the pusher has an equal structure. This connection of control model and device model can be seen as test bench. A path-time diagram for tongue and pusher in relation to the RUN and HALT commands is shown in Figure 7.

The simulation results are: The tongue is retracted from the carton before the pusher is retracted and thus the bag will remain in the carton. The RUN command starts the operation whereas the HALT command stops tongue and pusher after they have reached their initial positions.

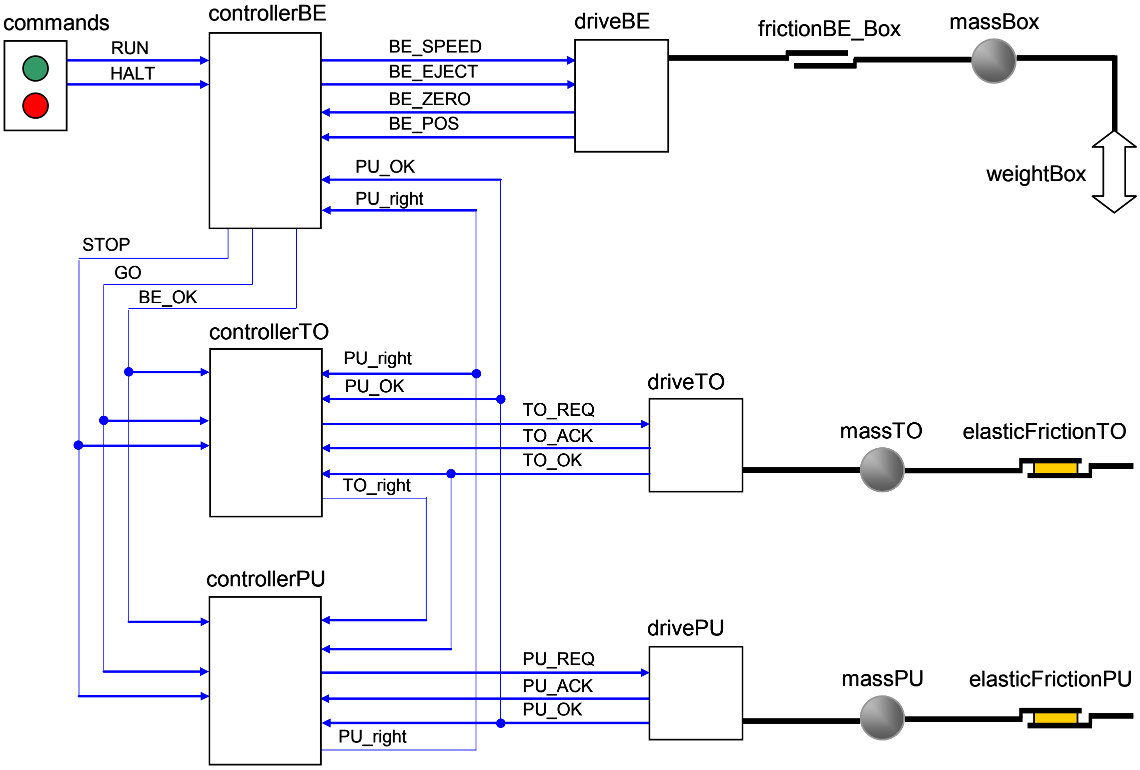

Figure 6: Structure of the case filling machine's overall system model

Secondly, the discharge belt's control is tested in its own test bench, shown in Figure 4. Thirdly, the models of tongue, pusher, and discharge belt are combined into an overall model of the case filling machine as depicted in Figure 6. The simulation results, shown in Figure 8, illustrate the belt position in relation to the movement of tongue and pusher as well as the value of the belt speed. The requirement that the belt moves only in the case that tongue and pusher are in the initial positions has been fulfilled. The HALT and RUN signals along with a filled carton result in a coordinated stop and restart of the course of events.

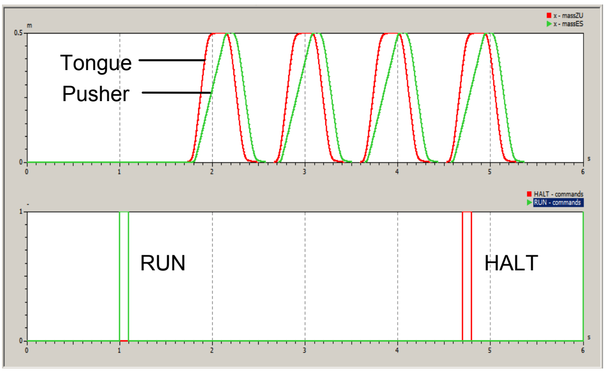

Figure 7: Path-time diagram of pusher and tongue

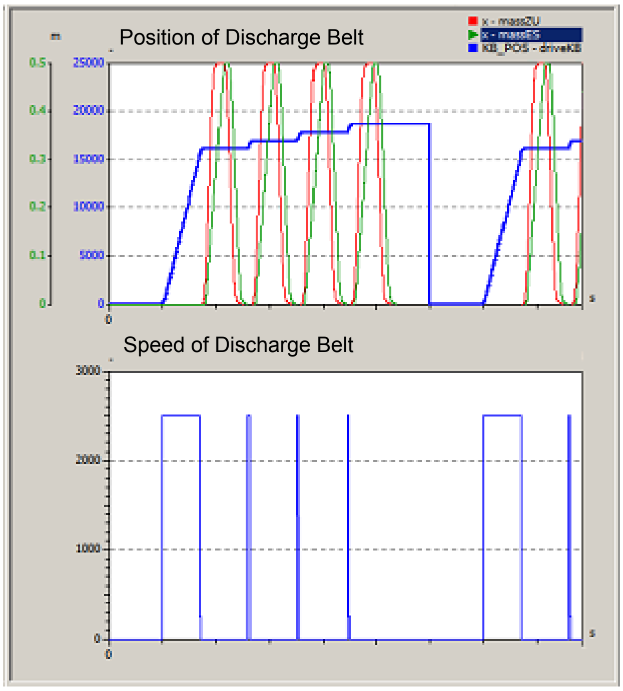

Figure 8: Simulation results of the overall model

4 Modelica and IEC 61131 code generation

4.1 Structure of the Modelica control program

The statecharts of the controllers of pusher, tongue, and belt are translated separately into Modelica code without any manual assistance. Figure 9 highlights the general code structure of these controllers.

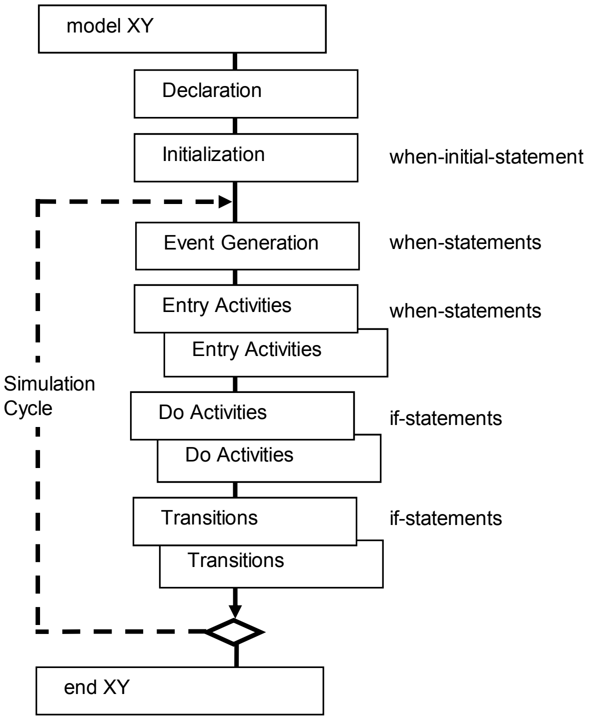

Figure 9: Structure of the statechart's Modelica code

The algorithm section contains the following blocks:

- Event generation:
  when-statements with relations of signals or times in order to toggle flags
- Entry activities:
  when-elsewhen-statements to check discrete state variables and enclosed
  signal assignments
- Do activities:
  if-elseif-statements to check discrete state variables and enclosed
  simple signal assignments
- Transitions:
  if-elseif-statements to check discrete state variables and to evaluate Boolean transition conditions and enclosed state assignments to the state variables and *signal assignments*.

The *Entry Activity Blocks*, *Do Activity Blocks* and *Transition Blocks* are created according to the hierarchy of the corresponding statechart. There is no hierarchy existent in the tongue's and pusher's controls (Figure 3). Hence they are translated into one *Entry Activity Block* and one *Transition Block* respectively. The belt's controls show a two level hierarchy which comprises the composite state Work and additional simple states (Figure 5). After code generation the two-level hierarchy is represented by two *Entry Activity Blocks* and two *Transition Blocks*.

4.2 Structure of the PLC program

The Modelica code's semantics are the standard for the code of the target controller. In this paper Structured Text (ST) according to IEC 61131 [7] is chosen as target code language. ST is analogue to Pascal and can be used to write programs for programmable logic controllers (PLCs).

The code sections in ST are similar to the statechart's Modelica code and can be separated into the *Event Generation, Entry Activities, Do Activities,* and *Transition* sections. In addition, the internal program structure requests a separation of declaration and executable functions.

I/O signals and global variables as well as the names of all implemented functions are noted in the symbol table. This table is created automatically when the target controller's code is generated [8]. Local data is defined and stored in a data block which is also part of the generated ST code.

The function *FC Event Generation (EG)* scans for signal events and sets or resets the corresponding flags. Because there is no *when*-statement in ST, edge detection is achieved by using additional flag variables and *if*-statements. An ST code example is shown below:

```
// Event Generation Block
IF NOT("DBSC1".trigEventFlag) AND ("in1">0)
THEN
    "DBSC1".trigEvent := TRUE;  //SignalEvent
ELSE
    "DBSC1".trigEvent := FALSE; //SignalEvent
END_IF;
IF ("in1">0) THEN
    "DBSC1".trigEventFlag := TRUE;  //EventFlag
ELSE
    "DBSC1".trigEventFlag := FALSE; //EventFlag
END_IF;
```

All entry activities of the states are contained in the function *FC Entry Activities (EA)*. A *case*-statement detects the active state according to the state variable. Thereafter the one-time activation of the entry activity is ensured by the implementation of a entry activity flag within an *if*-statement:

```
// Entry Activity Block of Main_State
CASE "DBSC1".S_1 OF        //StateVariable
3:     IF ("DBSC1".A_1 <> 3) THEN
        out1:=false;       //EntryActivity
        "DBSC1".A_1 := 3;  //EntryActivityFlag
       END_IF;
```

Do activities in are also implemented in the FC EA and noted similarly but do not contain an *if*-statement with an entry flag. Hence do activities are executed in contrast to entry activities in every PLC cycle.

The function *FC Transition (TR)* is comprised of transition triggers, exit activities and transition activities. As illustrated before the active state is detected in a case-statement. In each branch an if-statement exists which evaluates the corresponding transition trigger and holds the exit- and transition-activity. In case the transition's trigger condition is true, the exit- and transition-activity are executed and the state variable is set to the subsequent state:

```
// Transition Block of Main_State
CASE "DBSC1".S_1 OF
3:     IF ("trigger" > 0) THEN
        out1:=TRUE;         //ExitActivity
        out2:=FALSE;        //Trans.Activity
        "DBSC1".S_1:=4;     //State-Variable
       END_IF;
```

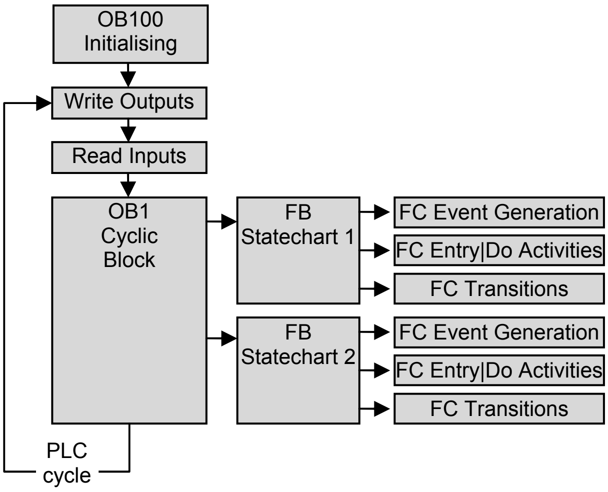

Figure 10: PLC program cycle

All aforementioned functions are called in the function block *FB Statechart* in the following sequence: *EG, EA* and *TR*. Timers which are used in the statecharts time triggers are also administrated in this function block. FB Statechart is called in organisation block OB1, which is the PLC's cyclic main program and is executed right after the update of the input register. The program of OB1 contains the calls of functions and function blocks which implement the control program functionality. After the execution of each function, the PLC jumps back into OB1 and when OB1's end is reached the current cycle ends with the update of the outputs. Figure 10 illustrates the PLC cycle and the call sequence of the control program's functions.

At startup of the PLC the initialisation block OB100 is executed which resets and initialises the state machine's internal variables. Organisation blocks OB1 and OB100 as well as function block *FB Statechart* are also part of the automatically generated ST program code. Hence a complete PLC program is generated from the controller's statechart.

After code generation two ASCII-files are available: one file containing the symbol table and one file containing the program code. Either file can be imported into the PLC engineering software Step7 by Siemens [9].

4.3 Compilation of the PLC program from a Structured Text source

The aforementioned transformation of a statechart into Structured Text indicates the similarities to Modelica, but there are additional requirements that need to be fulfilled. The main requirement which is caused by the PLC program's organisation into functions, function blocks, data blocks etc. says that within the ST source code called functions are stated in front of the functions that call them [10].

Therefore the order of the functions in the ST source code has to be as follows:

1. Data block *DB* with internal variables
2. Functions *FC Event Generation, FC Entry and Do Activities, FC Transitions*
3. Function block *FB Statechart*
4. Organisation block *OB1 Cyclic Block*
5. Organisation block *OB100 Initialising.*

The SCL batch compiler [10] of Step7 transforms the ST source code into MC7 code, which is executed by the PLC, and carries out a lexical, syntactical and semantic analysis and thereafter generates automatically all defined functions and blocks. They are stored in the design environment and can be loaded into as well as executed and tested on the PLC.

In case the automation systems controls are distributed over several statecharts, as is the case with the case filling machine described in section 2, then one set of source files (symbol table and ST code) is generated for each statechart. An add-on tool was implemented which merges all source files into one file set. The tool scans all symbol tables, eliminates double entries and copies all other entries into one table. The ST sources are also combined into one source. Functions and data block of the statecharts remain unchanged only their numbering is adjusted. The main task is to merge all organisation blocks (OB1's, OB100's) into one block of each type as there is only one instance of each type allowed in the PLC. At this point it is very important to consider the call sequence of the state machines on the PLC in order to avoid unwanted effects such as racing conditions caused by the serialisation of the PLC code [12].

5 Software-in-the-Loop tests

The simulation of the automation system can now be taken to a new level which will be illustrated in the following by using the belt controller as an example. After the PLC program has been generated, compiled and loaded a co-simulation of machine model (Modelica simulation) and PLC program (software PLC) can be conducted. The simulation tool PLCSIM [11], which is part of the Step7 PLC engineering system, is applied as software PLC. PLCSIM runs as a separate process on any Windows PC and enables the execution of the PLC program analogue to a hardware PLC. As a simulator PLCSIM offers no interface to hardware I/Os communicating with the field and thus can only be employed for simulation purposes.

In order to perform a co-simulation the inputs and outputs of the PLC have to be assigned to the corresponding signals of the machine model and cyclically updated in both directions.

The Modelica statechart model in the simulator has to be substituted with a coupling element that has exactly the same input and output signals as the statechart block. There are coupling elements available in the system simulator but these do not provide a suitable interface for a connection with the PLC. In addition the PLC's timing needs to be adapted to the system simulators timing or in other words, the PLC can no longer run in real-time but needs to run synchronous to the simulators simulation time.

Hence a coupling tool was designed and implemented which enables the coupling of PLC and system simulator. The tool provides interfaces in either direction. A TCP-socket connection is used for communication with the system simulator whereas communication with the PLC is achieved via a COM-object. The coupling tool, also called Backplane, is configured and then started by the user.

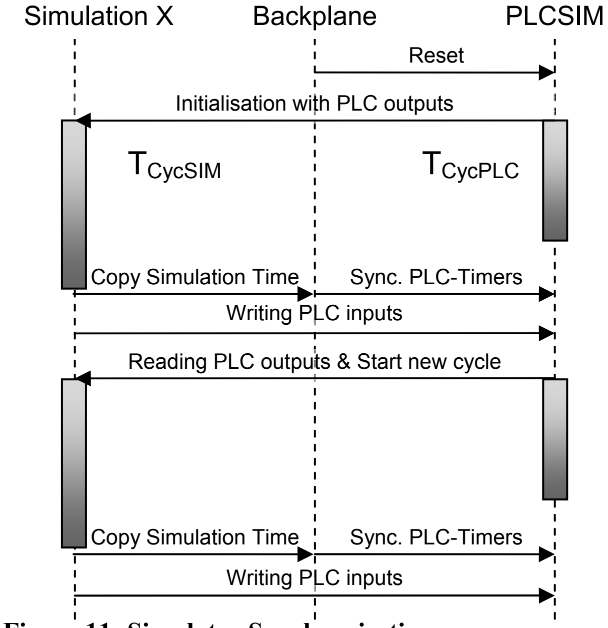

Figure 11: Simulator Synchronisation

Apart from configuration data such as IP-address and port, also the assignment of PLC I/O signals to the system simulator signals is defined.

In addition all PLC timers that are used somewhere in the PLC program have to be noted, as they will be synchronised by the Backplane. The Modelica simulation for operator and machine model is running in simulation time whereas the PLC is running in real-time. Therefore the synchronisation of time is indispensable and so the PLC's timers are synchronised at the beginning of each PLC cycle. Hence the progression of the timers is no longer related to the real-time, but depends only on the simulation time.

After the start of the simulation the Backplane takes over the control of the Modelica simulator and the PLC simulator and performs the data exchange between them. As show in Figure 11 in both simulators the same fixed time interval is simulated and thereafter the simulation is halted. The real-time duration T_{CycSIM} and T_{CycPLC} of the simulation cycles may differ. The Backplane scans the sensor signals of the machine model and writes them to the PLC's input signals. Then the PLC's output signals are read and transferred to the machine model's actuator signals before a new simulation cycle starts. This mode of operation is shown in Figure 11.

After the simulation run the results from the co-simulation are compared to the results of the Modelica-only simulation. A path-time diagram (discharge belt's position) of these simulation runs is shown in Figure 12. The waveforms are not identical as there is a growing time offset between characteristic points and also a spatial offset. Both effects can be traced back to the PLC's cycle time which is not present at the Modelica model level. The state machine of the controller model responds immediately to a change of the input signals. As soon as an input signal from the machine or operator model triggers a transition the simulation is halted. The Modelica code of the statechart block will then calculate the subsequent state and execute all exit activities, transition activities, and entry activities. This procedure is repeated until a stable state configuration is found, i.e. no further transition can fire. Therefore state transitions consume no simulation time and activities are instantly visible. The simulation is then started again and will run until the next event occurs. This behaviour is caused by the simulator's model of computation.

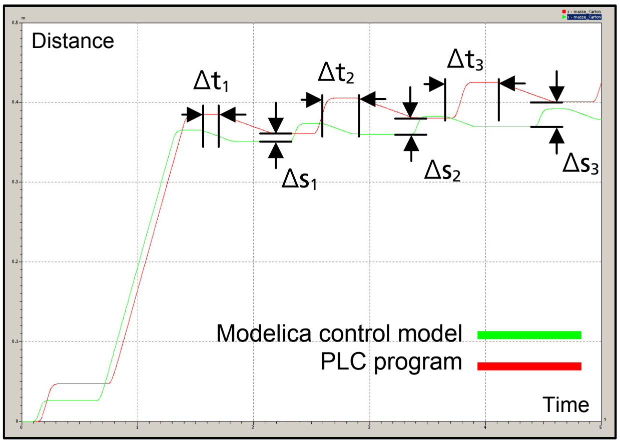

Figure 12: Path-time diagram of belt movement controlled by Modelica control model and PLC program

The PLC processes its control program after a different Execution Model [12]. As already discussed in section 4.2 the controller is executing the control program in a cyclic manner. Therefore the calculation of a new output signal vector requires one or more PLC cycles depending on the call sequence of the functions that where generated from the UML statechart. Typical cycle times for industrial controllers are 5 to 50 ms. The call sequence of the functions *EG, EA, TR* is also a cause for serialisation effects such as racing conditions. In case more than one state machine is used for controlling the system the sequence of their execution is also important. Such effects are not covered by the Modelica control model and can hence not easily be simulated. It is therefore necessary to extend the control model with an execution model of the real-time controller. At present the authors implement an execution model for PLCs at model level. The execution model is also designed as statechart and thereafter translated into Modelica code. First results of this approach are discussed in [12].

The main cause for the different behaviour of Modelica control model and real-time controller is the cycle time. For time-critical functions such as fast positioning operations a large cycle time can be unfavourable and result in the example of the case filling system in delays and spatial offsets when positioning the discharge belt as depicted in Figure 12.

The Backplane provides the option to force the cycle time. Therefore simulation runs with different cycle time can be simulated and the effects on the results analysed. This data can then be employed to find an upper limit of the cycle time which guarantees the correct and exact function of the system. The determined value is a criterion for the selection of a PLC.

The generated PLC program is tested against predefined test cases and the results are scanned for errors or deviations from the control models results. All test cases should encompass an extensive amount of input signal combinations and not only valid but also erroneous signal combinations. In case the control programs behaviour and function was correct the PLC program can be transferred to the real system without further adaptations.

6 Conclusion

In this paper the model-based design and simulation-based verification of an automation system as illustrated in Figure 13 was presented. Following the example of an industrial case filling machine the modelling of its control with statecharts was demonstrated. These statecharts were transformed automatically into Modelica code and then executed in test benches. For verifying the control algorithms the Modelica control blocks were stimulated with test signals and their behaviour was improved until the function corresponded to the defined requirements. Thereafter the Modelica blocks of control model, machine model and operator model were combined to an overall system model and a system simulation was executed by which the system's function was verified and important performance indicators were established. Once the behaviour and performance of the system was correct the IEC 61131 Structured Text code for the real-time controller, a PLC, was generated. The PLC's control program was then tested and validated as software-in-the-loop against the machine and operator model. This procedure showed a satisfying behavioural consistency between Modelica control model and real-time control program and only minor deviations were detected. These effects can be traced back to the PLC's cycle time. In order to overcome such deviations the authors are working on the implementation of an execution model for real-time controllers in Modelica.

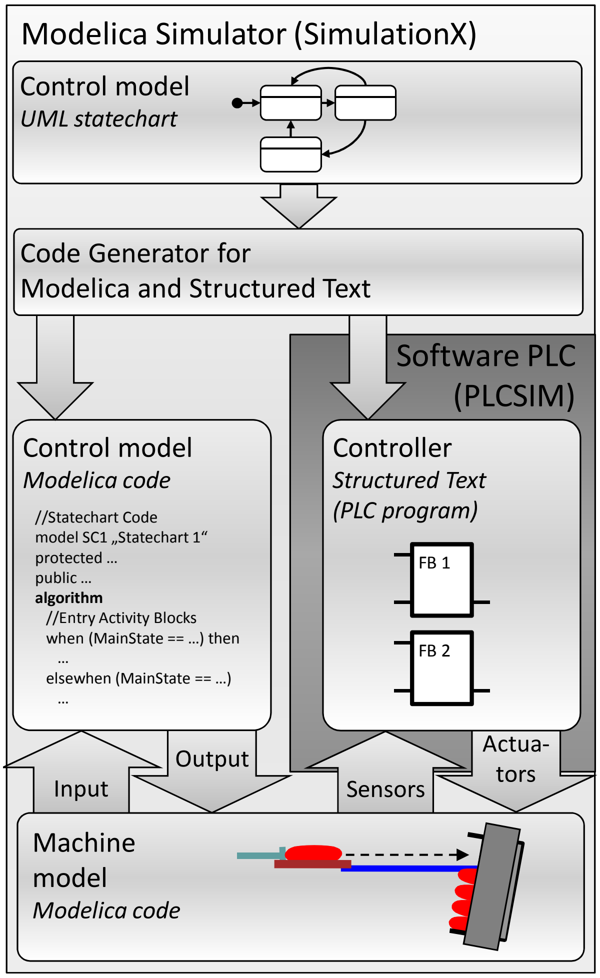

Figure 13: Design flow for modelling and verification of machine controls

The approach discussed in this paper removes the need for the error-prone manual coding of the PLC program and saves a huge amount of time by minimising the field and start-up tests through simulation-based verification of the control algorithms. The design-flow is faster and much more efficient than the current state-of-the-practice procedures and provides an easy way to error-free control programs through graphical and model-based design, simulation-based verification, and automatic code generation. The transformation steps in this chain are carried out automatically so that human efforts can be focused on the design of the controls and its evaluation.

References

[1] Schwabe, S.: Modellbasierter Systems-Engineering-Prozess. ECONOMIC ENGINEERING 3/2009, S. 58-59 http://www.berner-mattner.com/.../BernerMattner_Fachartikel_ModellbasSystemsEngProzess.pdf, visited on: 20.01.2011

[2] Haufe, J., Donath, U., Lantzsch, G.: Modellbasierter Entwurf von Steuerungen in der Automatisierungstechnik. Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS), Dresden, March 2009

[3] https://www.modelica.org/documents/ModelicaSpec30.pdf, visited on: 20.01.2011

[4] OMG Unified Modeling Language Superstructure V2.2

[5] Donath, U.; Haufe, J.; Blochwitz, T.; Neidhold, T.: A new approach for modeling and verification of discrete control components within a Modelica environment. Proceedings of the 6th Modelica Conference, Bielefeld, March 2008, p. 269-276

[6] http://www.simulationx.com, visited on: 20.01.2011

[7] Int. Electrotechnical Commission: IEC Standard 61131-3: Programmable controllers - Part 3, 1993

[8] Lindner L.: Rapid Control Prototyping by Transformation of Hierarchical State Machine Control Models into IEC 61131 PLC Code. Diploma thesis, TU Dresden, 2009.

[9] Siemens AG. Software for SIMATIC controllers. http://www.automation.siemens.com/mcms/automation/en/automation-systems/industrial-automation/Pages/Default.aspx, visited on 20.01.2011

[10] Siemens AG. S7-SCL V5.3 for S7-300/400 Manual, 2005

[11] Siemens AG. SIMATIC S7-PLCSIM V5.4 Manual, 2007.

[12] Seidel S., et al.: Modelling the Real-time Behaviour of Machine Controls Using UML Statecharts. Proceedings of the 15th International IEEE Conference on Emerging Technologies and Factory Automation, Bilbao, September 2010