49. Internationales Wissenschaftliches Kolloquium
Technische Universität Ilmenau
27.-30. September 2004

Holger Rath / Horst Salzwedel

ANSI C CODE SYNTHESIS FOR MLDESIGNER FINITE STATE MACHINES

Abstract

The complexity of systems using electronics increases rapidly, causing severe problems in the validation of this systems. Formal specification methods have to be included in the design flow, in order to achieve validation by design, to increase reliability, and to reduce development time and costs.

Finite state machines (FSMs) are an important specification methodology to describe reactive system processes, starting from system and mission level design to the partitioning layer of hardware/software co-design.

This paper describes automated ANSI C code synthesis for FSMs of the system design tool MLDesigner and shows, how automated code generation can be used to enable a seamless design flow. In section 1, the major semantic features of MLDesigner FSMs are introduced. An appropriate ANSI C code generator is described in section 2, including necessary tradeoffs as well as evaluation of standard implementation techniques. In section 3, the design of a controller for a LEGO® Mindstorms™ robot examplifies the usage of the implemented code generator. Finally, section 4 draws conclusion about achieved results.

1 Semantic Features

MLDesigner FSMs are mainly based on statecharts, developed by David Harel, supplemented by a special hierarchy concept and some tool-specific model elements.

For interaction with its environment, a state machine possesses input ports to receive external events and output ports to send new created events. On the basis of this interface, MLDesigner FSMs can be embedded at a higher modeling level into either a discret event or dataflow model.
Inside a state machine, states can be nested to an arbitrary depth. In opposition to statecharts, thereby only XOR-decomposition of states is supported. In MLDesigner, the AND-decomposition of concurrency models is decoupled from state machines and can be modeled outside in either a discret event or dataflow concurrency model of computation.
For each state machine, an initial state has to be defined for the top level as well as for each level of a state hierarchy. Graphically, an initial state is determined by small black circle with an arrow pointing to a state. All states without any child states are called leaf states. For each state, entry and exit actions can be defined, which are executed when the associated state is entered and left respectively.
Additionally, for each leaf state a so-called slave process can be defined, in the form of a MLDesigner module or another FSM. With respect to such a slave model, the superordinated state machine is called the master FSM. Compared to the direct XOR-decomposition of states, the master-slave-concept represents a second variant to split the function of a state machine hierarchically.
A special kind of states are the so-called history connectors. Inside a hierarchical state, a history connector remembers, depending on its definition, the last current sub-state at the

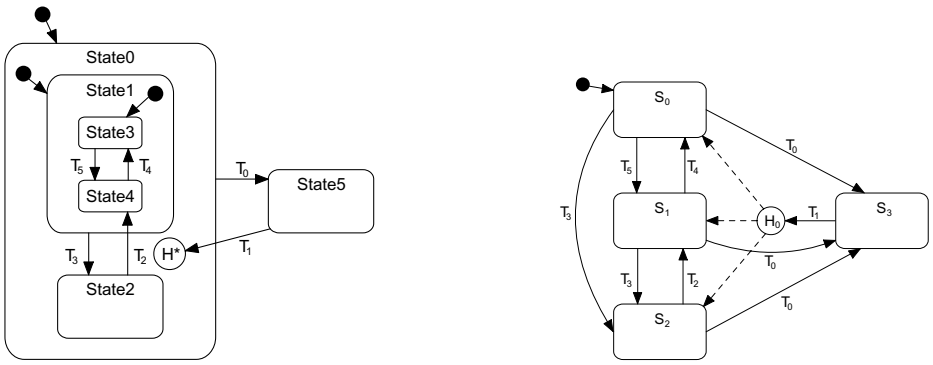
same hierarchy level (static history) or the last current leaf state at the lowest hierarchy level (recursive history).
State changes are described on the basis of transitions. In the form of a directed edge, a transition points from a source state to a target state. Child states inside a hierarchical state inherit all outgoing transitions of their parent state, whereby inherited transitions have a higher priority for firing. In addition to actions, which are executed during a state change, an *event expression* and a *guard condition* can be defined for each transition. The event expression specifies the events to trigger a state change. Additionally, the guard condition can be used to define event-independent state change conditions.
If the source state of a transition possesses a slave process, it has to be determined, if this transition is preemptive of the appropriate slave model or not. When a FSM received new events, first all outgoing preemptive transitions of the current state are checked for firing. If none of them is able, on the basis of its event expression and guard condition, to perform a state change, an existing slave model of the current state is executed. Afterwards all outgoing non-preemptive transitions of the current state are checked for firing [1, 2].

In addition to the standard elements of statecharts and the master-slave-concept, MLDesigner FSMs support all important tool-specific model elements. Parameters can be used to control a FSM functionality, whereby a parameter represents a constant value, which can be set individually in every FSM instance block. Memory elements can be used to define internal variables or to share external information with other MLDesigner modules. In addition to input ports, event elements can be used to indicate that a FSM can accept internal or external asynchronous events. All these elements, including the input and output ports, can be accessed by the actions associated with states and transitions [2, 4].

2 ANSI C Code Generation

On the basis of an integrated code generator, ANSI C code synthesis has been developed for single MLDesigner FSMs as well as for complete FSM modules, which consist at the lowest module level just of interacting state machines. Main application field of this code generator is the automatic creation of controller software for embedded and real-time systems. Since micro-controllers of such systems are generally characterized by limited memory resources and high performance conditions, some trade-offs had to be made between the semantic features of MLDesigner FSMs and the functional range of the code generator.
In this context, the code generator accepts only one hierarchy approach. Because XOR-decomposition of states, especially in connection with object-oriented concepts like generalization and specialization, is more important for modern design of reactive systems, the master-slave-concept is not supported. Another limitation has been made in connection with data types of specific FSM elements, like input/output ports, parameters and memory elements. To avoid critical memory fragmentation effects at run-time, the code generator accepts only data types of a fixed memory size. The set of these data types comprises integer and floating point numbers, enumerations and fixed sized character strings, integer vectors and floating point vectors. Additionally, composite data structures of these data types are supported. Before ANSI C code is generated for either a single state machine or FSM module, the code generator verifies the semantical correctness of all involved FSMs and checks, if the additional semantic limitations have been considered during modeling [2].

The most critical part of automated code synthesis for state machines is the representation of events, states and transitions. Standard implementation techniques, using a so-called *nested switch statement* or a *state table*, are not qualified for MLDesigner FSMs, because they are



(a) hierarchical (b) flat

Figure 1: Example FSM

mostly applicable to classical flat state machines. Furthermore, in the application field of embedded and real-time systems, these approaches cannot be used to implement FSMs with a larger number of states. The performance of a nested switch statement decreases with an increasing number of cases and the state table technique requires a large two dimensional array, which is typically sparse and wasteful. [2, 3].

The code generator for MLDesigner FSMs uses a new developed technique, which is able to implement hierarchical states, inclusive of history connectors. This technique has some parallels to the state table approach, but without any waste of memory resources.
First of all, hierarchical state machines have to be flatten, whereby only leaf states are considered as transition source states. In doing so, inherited transitions are multiplied and directly connected to leaf source states. History connectors are flatten by a list of all possible target states. Figure 1 shows an example hierarchical FSM and its appropriate flat representation. The internal identifiers of the flat state machine are listed in table 1.

Name	State3	State4	State2	State5	State1	State0	History in State0
Identifier	S_0	S_1	S_2	S_3	S_4	S_5	H_0

Table 1: Example Internal Identifiers

The core of this implementation technique is the so-called *state transition list (STL)*, determined by an one dimensional array of size n, where n is the number of possible state changes inside the flat state machine representation. Each array cell is a state transition function, defined as a pair of a state and an outgoing transition (S_i, T_j). A state transition function handles a complete state change, including performance of all associated state and transition actions. The complete list is sorted in two different ways. First, STL entries are ordered by their state index in terms of all possible state changes, starting from a specific state, represent a contiguous sublist. Secondly, within each sublist, state transition functions are sorted by their transition priority. Table 2 shows the state transition list for the example FSM in figure 1.

To ensure a fast FSM execution, the implementation technique defines two additional integer arrays of size m, where m is the number of states inside the flat state machine representation. For each state, the value of the *StartIdx* array determines the index of the first STL entry belonging to this state and the value of the *EndIdx* array refers to the last STL entry respectively. In other words, all possible state changes, starting from a state S_i and sorted

STL Index	0	1	2
STL Entry	(S_0, T_0)	(S_0, T_3)	(S_0, T_5)
STL Index	3	4	5
STL Entry	(S_1, T_0)	(S_1, T_3)	(S_1, T_4)
STL Index	6	7	8
STL Entry	(S_2, T_0)	(S_2, T_2)	(S_3, T_1)

Table 2: Example State Transition List

by transition priority, are given by the list:

$$STL_{Si} = (STL[StartIdx[S_i]], STL[StartIdx[S_i] + 1], \ldots, STL[EndIdx[S_i]]) \quad (1)$$

For the example FSM in figure 1, both arrays are defined as shown in table 3 and table 4.

Index	0	1	2	3
Value	0	3	6	8

Table 3: Example StartIdx Array

Index	0	1	2	3
Value	2	5	7	8

Table 4: Example EndIdx Array

Similar to the state transition list, each history connector is implemented on the basis of a so-called *history target list (HTL)*. This list is also defined as an one dimensional array of size k, where k is the number of possible target states of a particular history. HTL entries are history target functions, defined as pairs of a history connector and one of its target states (H_i, S_j). Table 5 describes the history target list of the example history connector.

HTL Index	0	1	2
HTL Entry	(H_0, S_0)	(H_0, S_1)	(H_0, S_2)

Table 5: Example History Target List

In addition to the implementation of FSMs, the developed code generator outputs a complete run-time environment, including a minimal real-time operating system (RTOS) and dynamic memory management. Principle task of this run-time environment is to control the execution of involved FSMs. In conjunction with real-time conditions, the generated run-time environment can be configured for a specific target platform [2].

3 Example

As an example for the usage of ANSI C code synthesis for MLDesigner FSMs, this section describes the design of a controller for a LEGO® Mindstorms™ robot, running on a Hitachi H8 micro-controller.

The underlying robot represents a machine to sort different colored blocks in reference to their color lightness. Thereby, only two cases are considered: dark and bright colored blocks. While the machine is running, blocks are moving one after another on a conveyor belt until a sensor has been reached and activated by the first block. In reaction, the conveyor belt stops and color lightness of this block is determined by a light sensor. The measured value is compared to a threshold and, based on the result, a sorter unit unloads the block to either left or right side. Once the sorter unit is ready again, the conveyor belt moves on.

Additionally, in case of interferences, a start/stop button is present to pause the procedure immediately and to continue it with a second press.

To develop the block sorter controller by using automated ANSI C code synthesis, first the complete function of the robot has to be modeled on the basis of MLDesigner FSMs. Since the robot mainly consists of a conveyor belt and a sorter unit, the control function is decomposed into a conveyor belt controlling FSM and a state machine, which handles the sorter unit. To recompose the control function, a Conveyor Belt FSM instance and a Sorter FSM instance are interconnected inside a higher level FSM module. This controller FSM module also defines the input/output interface of the controller.

Before the controller FSM module is used as input model for the code generator, it is validated by an appropriate simulation model, including a virtual 3D validation environment.

Figure 2 shows a MLDesigner screenshot with the 2 state machines, the controller FSM module and the appropriate simulation model. The virtual 3D validation environment and the real LEGO® Mindstorms™ block sorter robot are pictured in figure 3.

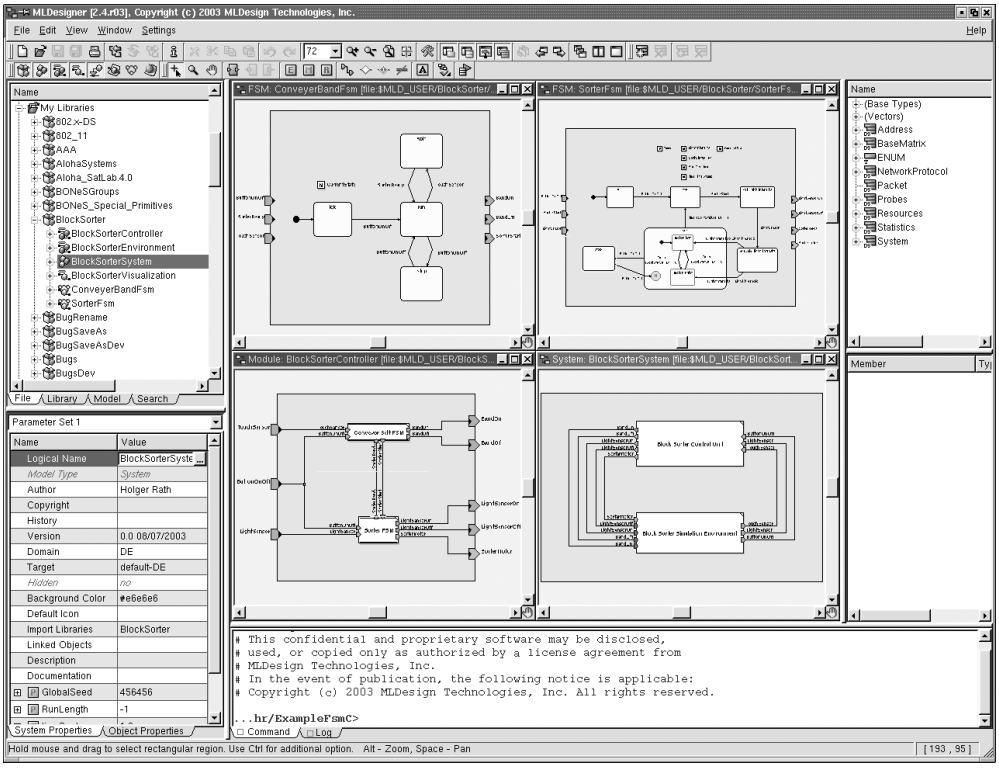
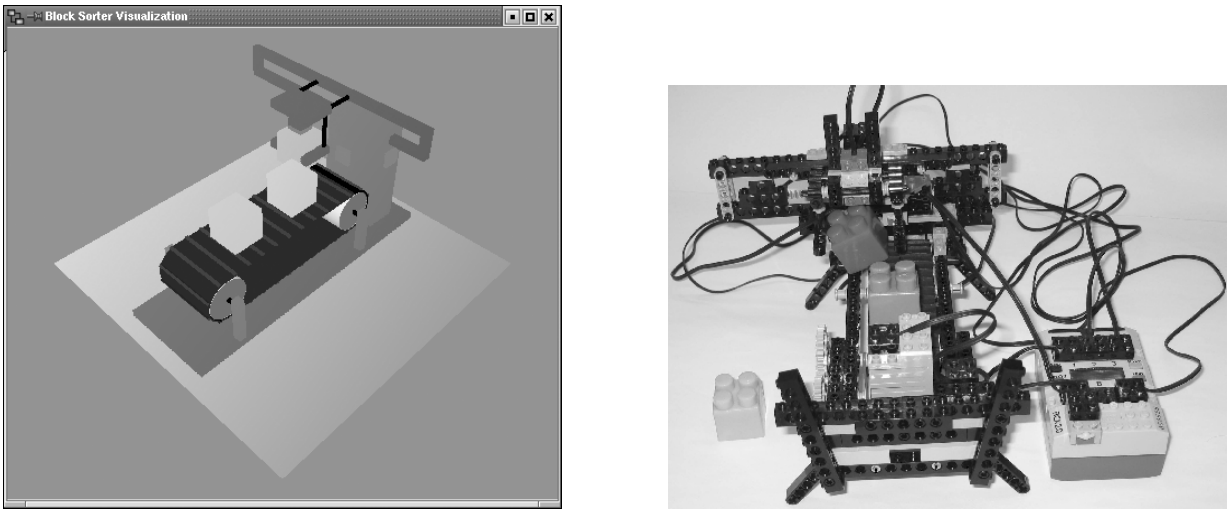


Figure 2: MLDesigner Block Sorter Models

After validation and automated ANSI C code generation, the actuator/sensor interface of the resulting code has to be configured using a C API for LEGO® Mindstorms™ motors and sensors. Finally, the compiled code can be uploaded to the Hitachi H8 micro-controller and the controller can be tested in interaction with the real motors and sensors.

4 Conclusion

Automated ANSI C code synthesis for MLDesigner FSMs, presented in this paper, fills the gap between high level design methodologies and low level application code for reactive systems. With respect to a few reasonable limitations, the developed code generator supports almost all semantic features. Since standard implementation techniques are



(a) virtual (b) real

Figure 3: Block Sorter Robot

not applicable to MLDesigner FSMs, a complete new implementation approach has been described. Finally, an example design has shown, how the implemented code generator can be used to enable a seamless design flow for a controller of an embedded system.

References

[1] Holger Rath, *Specification of the MLDesigner Finite State Machine Model*, Student research project, Ilmenau Technical University, 2002.

[2] Holger Rath, *ANSI C Code Synthesis of MLDesigner Finite State Machines*, Diploma thesis, Ilmenau Technical University, 2003.

[3] Miro Samek, *Pratical Statecharts in C/C++*, CMP Books, 2002.

[4] *MLDesigner User's Manual*, Version 2.4, http://www.mldesigner.com, 2004.

Author Information:

Dipl.-Inf. Holger Rath
Mission Level Design GmbH
Ehrenbergstrasse 11
98693 Ilmenau
Tel: (+49) 3677 / 4625-36
E-mail: holger.rath@mldesigner.de

Prof. Horst Salzwedel
Ilmenau Technical University
Faculty of Informatics and Automation
Department System and Control Theory
P.O. box 100565
98684 Ilmenau
Tel: (+49) 3677 / 69-1316
E-mail: horst.salzwedel@tu-ilmenau.de

Trademarks: