Design Models for Reusable and Reconfigurable State Machines

Christo Angelov, Krzysztof Sierszecki, Nicolae Marian

Mads Clausen Institute for Product Innovation, University of Southern Denmark,
Grundtvigs Alle 150, 6400 Soenderborg, Denmark
{angelov, ksi, nicolae}@mci.sdu.dk

Abstract. The widespread use of embedded systems mandates a rigorous engineering approach towards embedded software development, i.e. model-based design of embedded software. The paper presents design models of reusable and reconfigurable state machines that have been conceived in the context of the COMDES framework and in particular - the State Logic Controller and the Hybrid State Logic Controller, whose principles of operation are presented in the paper. The latter has been instrumental in developing a reconfigurable executable component, i.e. a function block of class State Machine, which can be used to implement a broad range of embedded applications such as sequential, continuous and hybrid control systems, as well as complex systems specified with hierarchical and concurrent state machines.

1 Introduction

State machines play an important role in embedded software design and recent research has focused on the efficient software implementation of state machines in terms of various types of design patterns [2, 7] and reconfigurable software components [3, 4, 9].

The conventional implementation of state machines is based on manual encoding of an abstract model such as the state transition graph, using 'switch-case' design patterns [7]. With this approach the state transition logic is built into the code, whereby a program has to be manually developed for each particular instance of the state machine. Consequently, such a program is difficult to modify and maintain. Moreover, this approach scales up badly with complex hierarchical and concurrent state machines. Therefore, a number of other design methods have been developed, e.g. The Quantum Framework [2] and the StateTable design pattern [7]. These methods provide design patterns that are potentially reusable at the source-code level but manual coding is not completely eliminated: the programmer is always required to code certain parts of the application (e.g. manual encoding of guards leading to selection of successor state and execution of related action). There are also some other problems, such as the use of sparse state transition tables resulting in considerable memory overhead, and in some cases – run-time creation of tables, which make such methods impractical for deeply embedded applications.

Industrial computer systems usually adopt a different approach, whereby a software state machine is implemented by modeling the structure of its hardware counterpart, i.e. making a program that computes the state transition logic functions and executes the actions that are associated with various states. In particular, this is how sequential control programs are implemented in programmable logic controllers [9].

In both cases, conventional design methods have a major shortcoming: the resulting implementation is not reusable, because the logic of the state machine is "hardwired" in the code. Specifically, this means that a separate program has to be developed for each particular application. That might not be a problem for small state machines but it is obviously a big problem with large state machines having tens or hundreds of states. In this case the software implementation of the state machine is a time-consuming and error-prone process, and the complexity of this problem rapidly grows with the number of states and state transitions.

The above problem can be eliminated through *reusable* implementation of state machines being based on re-configurable data structures and executable components such as discrete I/O drivers, state machine drivers, function blocks, etc. [3, 4]. The resulting software construction can be viewed as a higher-level object of type 'state machine'. That object might have multiple instances depending on the contents of the encapsulated data structures (configuration tables). The latter can be configured and re-configured via a dedicated configuration tool. In this way configuration of reusable components is substituted for conventional software design, and as a result of that manual coding of state machines can be potentially eliminated.

Configuration tables may contain information representing either state machine *structure* or state machine *behaviour*. In the former case the software implementation emulates the circuit diagram of the state machine (and indirectly – the behaviour of the state machine) by computing the corresponding state variables and the associated output signals for every invocation of the control program. Industrial controllers using this technique are known as *programmable logic controllers* (*PLCs*). In the latter case the software implementation emulates the behaviour of the state machine by directly interpreting a data structure representing the state transition graph or an equivalent behavioural model, e.g. a state machine flowchart. Industrial controllers using that technique are known as *state logic controllers* (*SLCs*).

The latter approach has a number of advantages: it is easier to use as it does not require preliminary logical design of the state machine, and it is ultimately simpler to implement. Therefore, this technique has been adopted while developing a reusable and reconfigurable state machine component in the context of COMDES - a software framework for component-based design of embedded control systems [1]. The related design issues are presented in this paper, which is structured as follows: Section 2 presents design models for reconfigurable state machines, such as the State Logic Controller and its enhanced version – the Hybrid State Logic Controller. Section 3 deals with the implementation of a reconfigurable function block of class State Machine. Section 4 presents related research. A summary of the proposed software design method and its implications is given in the concluding section of the paper.

2 Design Models for Reconfigurable and Reusable State Machines

The State Logic Controller (SLC) is built around a data structure, i.e. a state transition table that contains the computer representation of the state transition graph. This data structure can be efficiently implemented as a table containing modified (multiple-output) binary decision diagrams that represent the next-state mappings Fa_1, Fa_2, … , and so on, of various states within the state transition graph – the BDD Table (see Fig. 1 and Table 1).

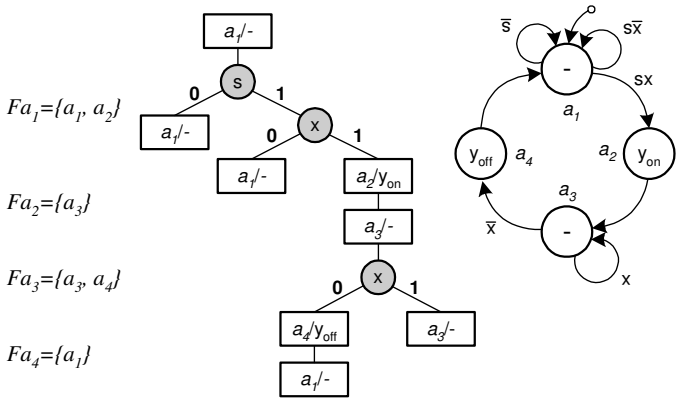

Fig. 1. Binary decision diagrams for next state mappings (Moore model)

Table 1. BDD Table of State Logic Controller for the example state machine

row	node	succTrue	succFalse	next-state mapping BDD
0	s	1	2	Fa_1
1	x	3	2	
2	a_1 / -	0	-	
3	a_2 / y_{on}	4	-	
4	a_3 / -	5	-	Fa_2
5	x	6	7	Fa_3
6	a_3 / -	5	-	
7	a_4 / y_{off}	8	-	
8	a_1 / -	0	-	Fa_4

This model is interpreted at run time, resulting in direct emulation of system behaviour: While operating, the SLC visits a number of states during successive control cycles *1, 2, 3,* … and executes the control actions associated with the visited states. These constitute a trace *P* in the state transition graph, e.g.:

$$P_i : \; a^i_1(1) \rightarrow a^i_2(2) \rightarrow a^i_3(3) \rightarrow \ldots , \qquad (1)$$

where a^i_1 is the initial (entry) state, $a^i_2 \in Fa^i_1$, $a^i_3 \in Fa^i_2$, etc. This type of behaviour can be illustrated by an execution trace of the example state machine shown in Fig. 1:

$$a_1 \rightarrow a_1(!s) \rightarrow a_1(s!x) \rightarrow a_2(sx) \rightarrow a_3 \rightarrow a_3(x) \rightarrow a_4(!x) \rightarrow a_1 \rightarrow \ldots \quad (2)$$

Such behaviour can be implemented with a standard application-independent routine - the *state machine driver* whose algorithm is given below assuming a *synchronous*, i.e. clock-driven Moore machine with binary (on/off) inputs and outputs:

```
State machine driver is: {
  // assuming knowledge of the last state visited during
  // the previous invocation
  do {
    determine current state from among the successors of the
    previous state;
      // by processing the next-state mapping BDD of that
      // state
    execute control action associated with current state
    (if non-empty);
      // read out corresponding word of control memory and
      // generate corresponding binary (on/off) signals
  } while (not(stable_state));
}
```

The binary decision diagrams of next-state mappings Fa_1, Fa_2, Fa_3, …. are usually processed one at a time, i.e. only one BDD is processed during the current cycle, which contains the successor states of the last state visited in the previous cycle. This mode of operation results in *step-wise* execution of control actions in successive time instants *1, 2, 3,* etc., which amounts to a strictly synchronous mode of operation. In practice, the above limitation can be relaxed in a number of ways, i.e. there are specific cases that introduce asynchrony, such as wait states of unspecified duration (e.g. states a_1 and a_3 in Fig. 1) and wait states of specified duration implemented with hardware or software timers. In these two cases some states may be revisited over and over again during a sequence of control cycles, which amounts to having states whose duration is a multiple of the basic clock period.

There is yet another case that introduces a different type of asynchrony, whereby a number of states may be visited in a sequence of *immediate transitions* carried out during the current control cycle. In this case the controller executes a sequence of operations (e.g. arithmetic and comparison operations) before reaching a *stable* state, where it breaks out of the sequence and exits.

The above discussion has been made assuming binary encoding of the control memory of the state machine. However, this assumption limits the presented software design to basic sequential control applications featuring binary input and output signals and predominantly synchronous mode of operation. In a more general context it might be necessary to implement additional functionality such as timers, event counters, arithmetic and comparison operators used to compute derivative condition variables, etc.

That can be accomplished by introducing lower-level software objects called *function blocks* (FBs) implementing the above functions, whereby function block instances may be invoked within separate states of the state machine. This model can be further generalized by making it possible to invoke not only individual function

blocks, but also function block sequences specified with function block diagrams, within a given state of the state machine. Such sequences are actually implemented as *composite function blocks* (*CFBs*). CFBs may also be used to compute condition variables needed for guard evaluation within the corresponding next-state mapping BDD. This extension results in executable hybrid models - hybrid state machines (see Fig. 2). The latter may be used to specify and implement a broad range of embedded applications, such as sequential control systems with analog input signals used to compute derivative condition variables, as well as continuous and hybrid (modal) control systems.

The first type of system is illustrated with the hybrid state machine shown in Fig. 3. In fact, this is the original example discussed above, which has been re-interpreted, so as to represent a hybrid state machine for a tank pressure control system. The latter observes tank pressure and switches on a discharge valve (y_{on}) when tank pressure exceeds a predefined limit value and conversely - switches off the valve (y_{off}) when pressure is normalized. In this example the input variable s is a binary signal generated by an input driver; the condition variable x is generated by a composite function block that must be invoked whenever the choice of successor state depends on that signal, and control signals y_{on} and y_{off} are generated by two instances of FB type Binary Control (*bCtrl1* and *bCtrl2*) executed in the corresponding states – a_2 and a_4.

The conceptual algorithm of the corresponding state machine driver is given below:

```
Hybrid state machine driver is: {
  // assuming knowledge of the last state visited during
  // the previous invocation
  do {
    execute condition function block(s);
      // basic or composite FB used to compute condition
      // (guard) variables labeling transitions from the
      // previous state to successor states, if necessary
    determine current state from among successor states;
      // by processing the next-state mapping BDD of the
      // previous state
    execute control function block;
      // basic or composite FB used to compute the control
      // action associated with current state, if non-empty
  } while (not(stable_state));
}
```

The conceptual design presented in this section has been used to implement a function block of class State Machine [1]. The main feature of this type of component is its ability to invoke instances of other (basic and/or composite) function blocks inside visited states. Ultimately, it is possible for a state machine to invoke another instance of the state machine while visiting a state (OR-decomposition). In that case each instance of the state machine function block is specified by a separate BDD Table. Thus, it is possible to recursively invoke the state machine driver while processing the tables of nested state machines. Likewise, *AND*-decomposition might be implemented by invoking a sequence of such function block instances that will be executed in an interleaved fashion (i.e. one after the other) within a superstate of the upper-level state machine.

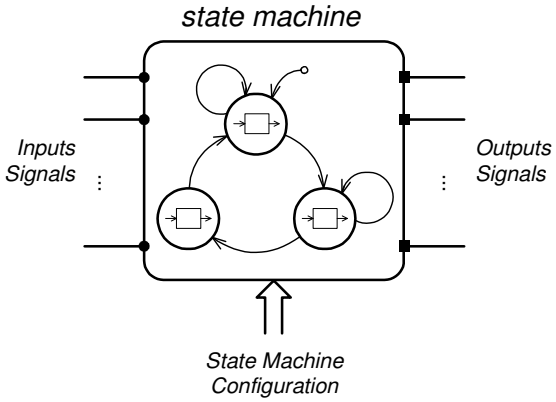

Fig. 2. Hybrid state machine

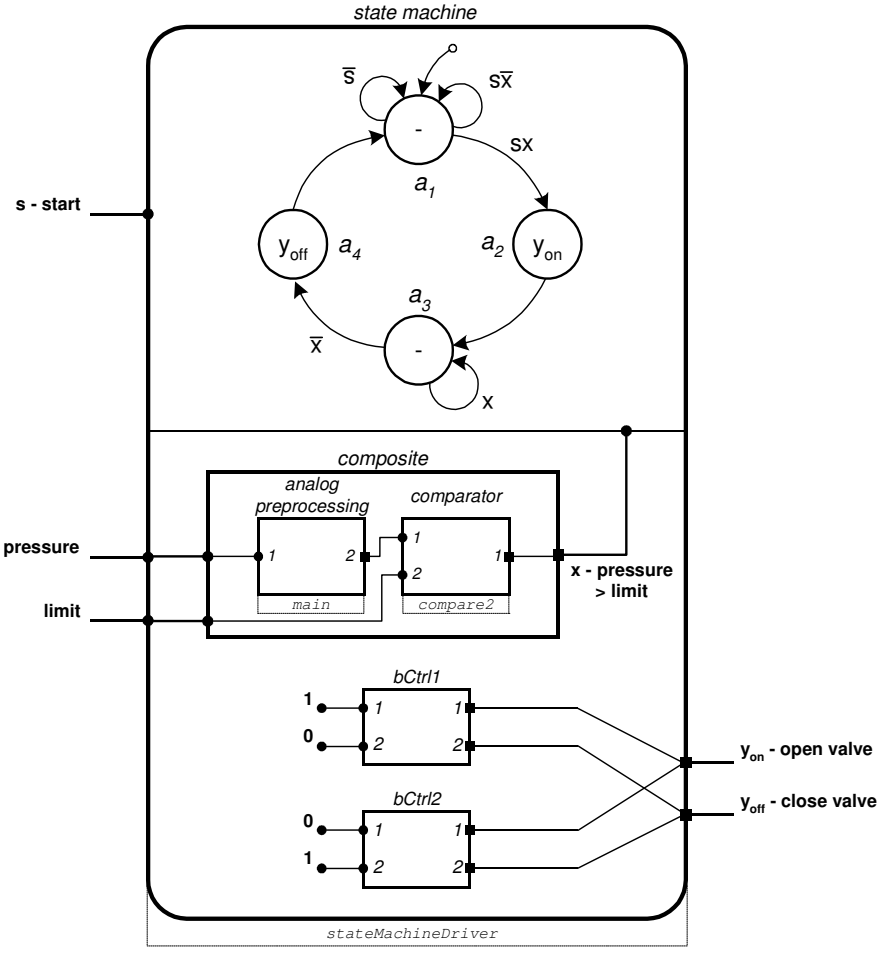

Fig. 3. Hybrid state machine for tank pressure control system

The above feature can be used to implement complex behaviours modelled by *hierarchical* and *concurrent* state machines [1]. In that case, during each invocation the program executes sequentially the constituent state machines until a *global stable state* is reached, and then breaks out until the next invocation, etc. A global stable state denotes a situation, whereby all constituent state machines have reached a stable state within the current step of execution.

Hierarchical composition of state machine function blocks provides a solution to the problems of complexity and scalability: a complex control system may be represented by a hierarchical model, whereby each constituent state machine is implemented with a different instance of that function block. An alternative approach is to decompose the object of control into functional subsystems, whereby each subsystem is assigned an individual controller modeled by a conventional (flat) state machine. In that case the overall control system may be viewed as a composition of interacting state machines that closely reflects the structure of the application [3].

3 Implementation Aspects

The algorithm given above outlines the principle of operation of the hybrid state machine driver. However, certain refinements to the algorithm need to be highlighted before discussing the actual implementation in more detail:

- The state machine driver interacts with two types of function block: function blocks that generate condition variables (*condition function blocks*) and function blocks that generate control signals (*control function blocks*). Both of them may be basic or composite function blocks (CFBs).
- The execution of condition function blocks is actually integrated with BDD processing, i.e. a FB is executed and its output signal is immediately used to make a branching decision before executing the next one, and so on, until a state node is reached,
- A condition function block might be executed only once if it generates several sequentially tested condition variables. Moreover, it is possible to test a number of such variables simultaneously via bit-patterns (mask and value) specifying the corresponding subsets of the tested variables and their values,
- The execution of condition function blocks is ultimately integrated with the execution of control function blocks, whereby BDD processing ends up with the selection of current state and the execution of a control function block associated with that state,
- It is possible to specify empty (Null) states, e.g. wait states.

It is assumed, that state transitions are complete and consistent. Completeness and consistency must be checked at configuration time, and this can be done using techniques similar to those presented in [8].

The above options have been taken into account when designing the data structures and the algorithm of the hybrid state logic controller, which are presented below. The discussion is illustrated with the example hybrid state machine given in Figs. 1 and 3.

The SLC algorithm parses a state transition table (STT), which consists of data records of the following structure:

```
typedef struct {
  TFBType        type;
  TFBFunction    function;
  TFBInstance    instance;
  TConditionVar* conditionVar;
  TConditionVar  mask;
  TConditionVar  value;
  union {
    TSTTRow      successorTrue;
    TSTTRow      nextState;
  };
  union {
    TSTTRow      successorFalse;
    TBool        immediateTransition;
  };
} TSTTRecord;
```

The records are grouped in segments representing the next-state mappings of various states of the state machine. A brief description of record fields is given in Table 2, whereby the function block is specified by three fields: *type*, *function*, and *instance*. These are used to invoke a function of a specified type on a given instance. The evaluation of transition guards to either True or False is based on the assessment of condition variables, via the *conditionVar* pointer, masked by *mask* field, and compared to required *value*. The execution of a control FB in a state is distinguished from the computation of condition variables by assigning Null to the *conditionVar* field of the STT record. Moreover, in the case of state node, *mask* and *value* fields are not used, and the meaning of the last two fields is *nextState* and *immediateTransition*. In case of transition guards evaluation all fields are used and the meaning of the last two fields is *successorTrue* and *successorFalse*.

Table 2. Description of state transition table record fields

Field Name	Field Description
type	Type of function block to be executed, index of function block type table: FBTypes[type]
function	Routine of the FB to be executed, index of function block routine: FBTypes[]→FBFunctions[function]
instance	Pointer to the FB instance execution record, argument passed to function block routine: FBTypes[]→FBFunctions[](instance)
conditionVar	Pointer to condition variable used in order to access results of function block computation
mask	AND mask imposed on condition variable

value	Value of expected masked condition variable used in order to evaluate transition guards either to True or False
successorTrue / nextState	Index pointing to successor row in case of transition guard variable evaluated to True / Index to the initial row of next state mapping in case of control action execution (state execution)
successorFalse / iTransition	Index pointing to successor row in case of transition guards evaluated to False / Flag indicating immediate transition to *nextState* in case of control action execution (state execution)

The implementation of the hybrid state machine driver is presented below:

```
 1  row = tableRow;
 2  do {
 3    if (row->instance != NULL)
 4      FBTypes[row->type]
 5       ->FBFunctions[row->function](row->instance);
 6
 7    if (row->conditionVar != NULL) {
 8      if ( (*row->conditionVar & row->mask)
 9                                    == row->value) {
10        row = row->successorTrue;
11      }
12      else {
13        row = row->successorFalse;
14      }
15    }
16    else {
17      tableRow = row->nextState;
18      if (row->immediateTransition == FALSE) {
19        return;
20      }
21      else {
22        row = tableRow;
23      }
24    }
25  } while(TRUE);
```

STT parsing is started from the row previously saved in the *tableRow* variable (line 1). Then, the algorithm will loop until a stable state is reached (line 2-25). In the loop, first the function block routine is executed if the *instance* is specified (line 3-5). It is possible to parse the STT without executing a function block, e.g. an empty state, or a transition guard evaluation based on condition variables computed earlier (see example below). If the *conditionVar* field is not Null, evaluation of a guard variable takes place, and a successor row is chosen (line 7-15). When the *conditionVar* field is Null, the control function block associated with state has been just executed, and a

state has been reached (line 16-24). If the *immediateTransition* field is equal to False a stable state has been reached and the SLC leaves the loop (line 18-20), otherwise it continues looping (line 21-23).

The presented design pattern will be illustrated with the example state machine shown in Fig. 1 and Fig. 3. Table 3 shows the encoding of the state transition table for that example, where grey rows represent condition nodes and white rows - state nodes of the BDD shown on Fig. 1.

Table 3. State transition table of example state machine

row	cond. var. / state	type	function	instance	condVar / state	mask	value	succTrue / nextState	succFalse / iTrans	next-state mapping
0	s	-	-	Null	&inpDriver.s	1	1	1	2	
1	x	TCFB	fBD	&composite	&composite.x	1	1	3	2	Fa_1
2	a_1	-	-	Null	Null	-	-	0	False	Fa_4
3	a_2	TBCtrl	main	&bCtrl1	Null	-	-	4	True	
4	a_3	-	-	Null	Null	-	-	5	False	Fa_2
5	x	TCFB	fBD	&composite	&composite.x	1	1	4	6	Fa_3
6	a_4	TBCtrl	main	&bCtrl2	Null	-	-	2	True	

It can be seen from the table that state nodes are differentiated by means of Null *condVar* pointers. This is because state nodes are associated with control function blocks, e.g. a_2 and a_4, or may represent empty states having a Null *instance* (e.g. a_1). Condition nodes are associated with condition function blocks, whose output location is accessed by means of a *condVar* pointer, and the corresponding output bit(s) are tested using the specified *mask* and *value* fields. A condition node may have a Null *instance* if the corresponding function block has been previously executed (e.g. the input driver FB generating signal *s*).

Rows are usually grouped in contiguous segments representing the next-state mappings (i.e. subsets of successor states) for the corresponding states of the state machine. However, the table can be eventually minimized taking into account that some of those mappings may be subsets of other next-state mappings. Here, $Fa_2 \subset Fa_3$, *and* $Fa_4 \subset Fa_1$, which is reflected in the composition of the example state transition table (see also Fig. 1).

When invoked, the state machine driver processes the table segment corresponding to the next-state mapping of the previous state (e.g. Fa_1) in order to select the current state (a_1 or a_2), execute the associated control function block and exits (if no immediate transition has been specified). During the next invocation the driver processes the segment containing the successors of the previously chosen state (e.g. Fa_2), and so on. However, in case of immediate transitions, the driver may process two or more segments in succession before a stable state is reached and the program is exited (e.g. state a_2 which is immediately followed by the stable state a_3, and likewise a_4, which is immediately followed by the stable state a_1).

4 Related Research

The increasing complexity of embedded applications has stimulated the investigation of hybrid models, e.g. mode-automata implemented in LUSTRE [5] and the hybrid models combining state machine and data flow domains in the Ptolemy II framework [6], which are similar to the hybrid state machine presented in the paper (Fig. 2). However, the above models are not implemented as reusable and reconfigurable components.

The importance of reconfigurable components, and in particular - reconfigurable state machines has been recognized, and there are already several research projects and industrial developments illustrating this approach, e.g. StateWORKS [3]. The latter employs 'virtual' finite state machines (VFSMs), which use state transition tables that are interpreted at run time by a standard routine called the VFSM executor. The system uses a non-hierarchical event-driven state machine model with a combined Moore/Mealy semantics. Consequently, an application is conceived as a hierarchy of flat state machines, which can be (re)configured by generating the necessary state transition tables. A similar approach has been developed for open machine control systems [4], where reconfigurable state machine components are once again implemented by means of state transition tables interpreted at run time by a state machine driver. This architecture uses a purely event-driven Mealy model (i.e. no guards are specified in the state transition table) and it supports hierarchical state machines in a Statecharts-like fashion.

However, these two systems do not support function blocks as defined in IEC 61131-3 and similar industrial standards [9]. Instead, the StateWORKS environment uses the concept of virtual inputs/outputs provided by a virtual I/O processor (also denoted as real-time data base). The latter consists of objects that are instances of predefined or user-supplied object classes. Likewise, the open machine control architecture employs user-supplied functions invoked from within the state machine driver in order to execute output actions. Another difference is in the data structures used to implement the state transition tables. These are relatively complex in both cases, resulting in processing and memory overhead, which can be substantially reduced by using BDD-based data structures, as suggested in this paper.

The presented function block model (Fig. 2) bears certain resemblance to function blocks defined in standard IEC 61499, which also incorporates an execution control state machine. However, that state machine is "hardwired" in the function block, i.e. it uses predefined sets of inputs and outputs and its configuration cannot be changed without reprogramming.

5 Conclusion

The paper has presented design models of reconfigurable state machines that have been conceived in the context of the COMDES framework, and in particular - the state logic controller and the extended (hybrid) state logic controller, whose principles of operation are presented in the paper. The latter has been used to develop a generic reusable component, i.e. a function block of class State Machine, which can be used

to implement a broad range of embedded applications, such as sequential, continuous and hybrid control systems.

The main idea of the proposed method is to provide a universal *executable* component that can be reconfigured *without any re-programming*. This is accomplished by updating the supporting data structure, i.e. a state transition table, whereas the executable code remains unchanged and may be stored in permanent memory. The state transition table consists of multiple-output binary decision diagrams (BDDs) that represent the next-state mappings of various states and the associated control actions. This solution has important implications:

- BDDs allow for compact encoding of state transition tables resulting in considerable memory savings in comparison with other design methods,
- BDDs allow for extremely fast processing of state transition tables via guided execution/testing of condition function blocks while evaluating a single guard specifying a transition to a successor state, thus avoiding the need to compute multiple guards in order to determine the successor state,
- This has also safety implications as long as BDD processing always results in the selection of a successor state, whereas other methods are prone to software errors that may result from incomplete and inconsistent specification of state transitions.

The presented function block design method has been validated in a number of real-time control experiments and most notably – a modal control system for a DC motor (speed and direction control) and a sequential control system for a complex plant specified in the Production Cell case study [5]. A configuration and analysis tool supporting the method is now under development.

References

1. Angelov, C., Sierszecki, K.: A Software Framework for Component-Based Embedded Applications. Proc. of the Asia-Pacific Software Engineering Conference APSEC'2004, Busan, Korea (2004)
2. Samek, M.: Practical Statecharts in C/C++: Quantum Programming for Embedded Systems. CMP Books (2002)
3. Wagner, F., Wolstenholme, P.: Modeling and Building Reliable, Re-usable Software. Proc. of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Huntsville, USA (2003)
4. Wang, S., Shin, K. G.: Constructing Reconfigurable Software for Machine Control Systems. IEEE Trans. on Robotics and Automation, vol. 18, No 4, August (2002), 475-486
5. Maraninchi, F., Remond, Y.: Applying Formal Methods to Industrial Cases: the Language Approach (The Production-Cell and Mode-Automata). Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems, Berlin (2000)
6. Lee, E.: Embedded Software – an Agenda for Research. UCB ERL Memorandum M99/63, University of California at Berkeley, USA (1999)
7. Douglass, B., P.: Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison Wesley (1998)
8. Heimdahl, M. P. E., Leveson, N.G.: Completeness and Consistency Analysis of State-Based Requirements. IEEE Transactions on Software Engineering, TSE 22(6), (1996), 363-377
9. John, K-H., Tiegelkamp, M.: IEC61131-3: Programming Industrial Automation Systems. Springer (2001)