EMBEDDED SOFTWARE DEVELOPMENT FOR MECHATRONIC SYSTEMS

Nicolae MARIAN
"Mads Clausen Institute for Product Innovation" University of Southern Denmark
Grundtvigs Alle 150, Dk-6400, Sønderborg, Denmark
nicolae@mci.sdu.dk

Abstract. *The widespread use of mechatronic systems mandates a rigorous engineering approach towards embedded software development, i.e. model based development using repositories of prefabricated software components. The main problem that has to be addressed in this context is to systematically develop the software framework for mechatronic applications, taking into account the true nature of them, which are predominantly continuous-time/discrete-event processes, hierarchical and concurrent systems. The implications of the adopted modelling technique by integration of component-based design and verification are discussed in the context of mechatronic domain systems. The result is a hierarchical-modular signal flow metamodel in which process and control tasks are clearly differentiated. The results have been used to formulate the guidelines used to develop COMDES - a software framework for distributed embedded applications.*
Keywords: *embedded software, model-based design, component-based design, reusable components.*

1. Introduction

In the industrial marketplace, the constant demand of ever greater functionality and reliability at ever lower prices and with shorter development cycles results in mechatronic products that are ever more complex, posing a serious challenge in design due to diverse, severe and conflicting requirements, and then, in verifying their correctness [2,5,6]. For many mechatronic devices, this complexity has remained manageable only because of the advent of the microprocessor: the economics and power of digital computation makes it the medium of choice for controlling systems composed of electro-mechanical and computational elements. In the first few prototyping cycles, the hardware may change significantly each time, further complicating concurrent software development. Different teams work on different tasks around the same product (e.g., control software, user interfaces, customer support, service documentation), often starting from little else than the semi-formal descriptions of the product. Such work practices are unable to deal with the increasing demand for faster time to market, and for greater flexibility in product lines. Faster development cycles essentially demand less hardware prototypes and faster, concurrent software development.

Traditional embedded software development methods involve production by hand using programming languages such as C or even assembly language, based on some available automated tools to help in the design. Most tools for embedded software are rather primitive compared to equivalent ones for richer platforms. When embedded software was simple, there was little need for a sophisticated approach, which is no more the case nowadays.

Another problem in development of embedded systems is that the implementation step often breaks consistency between the design models and the actual application, as the design models do not fit the implementation platform because the designer was simply unaware of platform specifics or left these out to simplify the design or enable certain verification methods. During the evolution of the system, the implementation and the design models often tend to get out of sync.

Some obvious response to these drawbacks were object-oriented approaches and other syntactically driven methods, but one that emerged and start to achieve best results is the model-based design using tools and generic “plug-and-play” components, using techniques for modelling continuous-time/discrete-event

and discrete/continuous values processes, hierarchical and concurrent systems [2,5]. These prefabricated components may be selected and aggregated by the customer, perhaps for the first time at the customer's location. However, this requires the development of software that is generic, reusable and customizable, rather than software that is custom-tailored for a particular configuration.

Component-based development of embedded software [1,2] is a change of focus from algorithmic and data structure related issues to the overall architecture of a software system, where the architecture is meant to be a collection of objects together with a description of their interconnection topology. A better structure can bring predictable and guaranteed behaviour under hard real-time constraints; scalable and open architecture supporting both design and analysis reuse.

The structure of the whole software system mirrors the architecture of the control process itself, which is made up of a number of independent modules, with a top-level hierarchical and concurrent controller for discrete inputs, events and conditions, and low-level prefabricated components for the continuous inputs, computational activities, and generated outputs. The model can be run step-by-step (simulation) or can be exposed to formal verification at all levels.

This paper defines, discusses, and illustrates a model-based design technique for embedded software systems. The advantages of the method are the well structuredness of the design, and the reusability of parts of these in relation to reuse of components, like in industrial software standards such as IEC 61131-3 and IEC 61499.

The Software Engineering Group of the Mads Clausen Institute for Product Innovation (MCI) has developed a number of integrated solutions to design and verification of embedded software systems under the name of COMDES (COMponent-based Software for Distributed Embedded Systems), in an attempt to solve the outlined problem.

The ultimate goal of our effort is to create an integrated embedded software development environment, applicable to most of the mechatronic systems, consisting of software configuration and analysis graphical tools that seems rich enough to configure and reason about different abstractions made for design and verification purposes using different semantic models through standardized prefabricated proved components, much in the same way as that is done in mature areas of engineering such as Mechanical Engineering and Electronics, and ultimately to efficiently produce reliable embedded real-time software.

The paper is structured as follows: Section 2 presents the modelling technique used to specify system structure and behaviour, i.e. the discrete part as composition and hierarchy of state machines, as well as the continuous part using simple or aggregated function blocks. An example of applying the method to a gripper system, consisting of an arm with shear and normal force sensors, a DC motor for gripping control, an actuator for lifting control, is sketched in Section 3. Section 4 presents main directions for verification of functional and timing behaviour of such systems, and application to our example. A summary of the software design method and its implications is given in the concluding section of the paper.

2. Component-based design in COMDES

Under COMDES, a distributed embedded application is specified in terms of interacting subsystems (function units), such as sensor, control unit, actuator, operator station, etc., as a set of state based, or not, signal flows of the form stimulus $\Rightarrow$ response [2]. Function units (*FU*) are defined as software logical integrated circuits encapsulating one or more dynamically scheduled tasks (activities) that are configured from state machines (*SMs*) (reconfigurable) and prefabricated lower-level components such as function blocks (*FBs*). The concurrent composition of interacting subsystems is synchronous in the way adopted by the synchronous model and languages, i.e. every subsystem that can make a transition upon its inputs in an execution instance, does so, and the reaction is considered instantaneously related to

changes, allowing recording all external events in the proper order [8].

2.1 COMDES components

COMDES software components are derived systematically from the engineering domain and they mimic hardware/control counterparts. The hierarchy of components is presented in Figure 1.

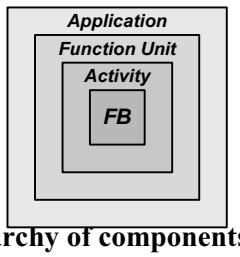

Figure 1. Hierarchy of components in COMDES.

2.1.1 Function Unit

Overall system configuration can be described by a function unit diagram, i.e. a signal flow graph specifying *FUs* and their interactions. *FUs* interact by exchanging signals, i.e. labelled messages (pressure, temperature, etc.) within various types of distributed transactions, such as producer-consumer, client-server protocols. Producer-consumer interaction is preferred for time-critical communication, because is one-to-many in the general case, and. it may have state-message or event-message semantics.

FUs encapsulate input/output signal drivers (the triangles in Figure 2), and one or more threads of control – activities. These generate application-specific reactions to timing and/or external events by invoking lower-level components - *FBs*, which implement specific signal processing and control functions. Activities interact with the outside world via signals provided by input and output signal drivers (by analogy with hardware integrated circuits).

The input/output drivers are in charge of decoding/encoding messages. COMDES framework uses content-oriented message exchanging, i.e. the message needs to be processed and then goes to the activities that are interested in it decomposed in internal signals. For example, the three PID control parameters K_P, K_D, K_I can be composed into one message by output drivers of the *OS FU*, and then the corresponding *FU* input driver will decompose it and distribute to different activity consumers in the *Controller FU*.

2.1.2 Activity

Activity encapsulates one or many *FBs* that perform different functions. Complex activity behaviour can be formally specified in terms of hybrid *SMs* modelled by hierarchical and concurrent *SMs*. An activity is aggregated by pre- and post-processing *FBs* and a Moore *SM*. Each state is associated with a specific reaction which is accomplished with one or more output signals that are generated in response to the corresponding event. A transition between states usually includes:

- *Trigger*: Activating event
- *Guard*: Predicate that must evaluate to true at the instant the transition is triggered
- *Effects*: Specifies an *Action Sequence* to be performed when the transition fires.
- *Source*: The source state
- *Target*: The target state that is designated after the transition is fired

A signal is computed via a sequence of transformations that can be modelled with a *FB* diagram, i.e. an acyclic signal flow graph whose nodes implement basic application functions such as PID.

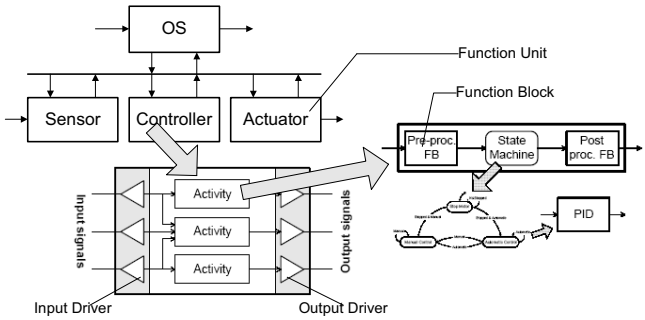

Figure 2. Generic objects of COMDES Framework.

2.1.3 Function Block

FB is the main COMDES component. *FB* can be seen as a class specifying a number of re-entrant and re-locatable routines, as well as a general definition of the so called *FB* execution record.

The latter is a data structure containing all necessary information such as parameters, signal inputs and outputs, internal variables, and is instantiated for each object of a given *FB* class.
A *FB* is ready to execute whenever the necessary signals arrive at its inputs, Figure 3. Then the basic *FB* (*BFB*) will execute, and produce signals at its outputs, which can become the input signals of the successor *BFB* and so on, until the last one is executed and the final output signal is generated.

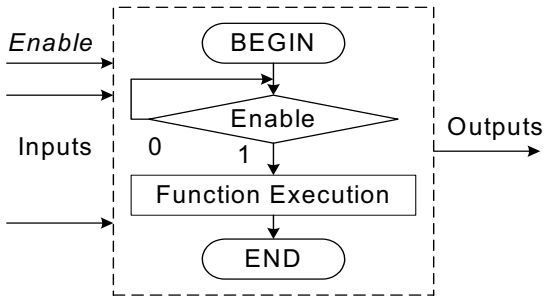

Figure 3. Control flow inside a *FB.*

FB instances are interconnected via softwiring using pointers to the corresponding data locations. Softwiring is conceived as an output-to-input(s) connection: output data is stored in the output buffer of the source *FB* instance record, and it is subsequently accessed by one or more destination *FBs* through the corresponding input pointers. This allows for efficient one-to-many connections by limiting the need to copy source data to multiple destination inputs.
Based on different interaction mechanism, softwiring can be classified to be *static* and *dynamic*. In the former case a *FB* input is connected to the output of another *FB* by means of an assignment statement:

```
FB_Instance1.input=FB_Instance2.out
                  put
```

It is obvious that this technique only support special-purpose applications. Dynamic softwiring can be implemented by the use of pointers, which point to corresponding data locations instead of direct references:

```
ADD_Table[0].input1_pointer =
      &(ADC_Table[0].output)
```

We can easily figure out that dynamic softwiring support both *one-to-one* and *one-to-many* interaction. And connections can be reconfigured by updating the values of input pointers in the corresponding instance records, without changing the code of software components involved, which is suitable for on-line reconfiguration.
Two or more *BFBs* that aggregate a complex function can be seen as a *composite FB* (*CFB*). For example, we can specify a *CFB* called *signal processing*, which is composed of *BFB* such as ADC, unit conversion, limit checking, etc
A *CFB* is externally indistinguishable from a *BFB*. It can be viewed as an aggregation of *BFBs* that execute in certain precedence by a standard routine called *FB driver*. The *FB* driver is a static *FB* execution scheduler that linearly invokes encapsulated *BFB* instances according to the execution schedule. The execution schedule is derived from the signal flow diagram depending on applications.

```
Function Block Driver: {
 i = 0;
 do {
  read SCHEDULE[i];
  execute control action specified by
  SCHEDULE [i].FB_type and
  SCHEDULE [i].FB_instance;
  i = i+1;
 } while (SCHEDULE[i].FB_type != NULL);
}
```

2.1.4 Input and Output Drivers

Input and output drivers are special types of components, data-oriented components, with peculiar features with respect to their I/O connection patterns:

- An input driver has no input connections to other *FBs*, i.e. it has physical inputs and one or more outputs (one place buffers) that are connected to the inputs of other *FBs*.
- An output driver has no output connection to other *FBs*, i.e. it has physical outputs and one or more inputs that are connected to the output buffers (again one place buffers) of the corresponding source *FBs*.

Input and output drivers are similar to *FBs*, but they are not inherited from the class of *FB*. They are special components aggregated in *FUs*.

3. Gripper system

The gripper system consists of a robotic arm with shear force sensors, microcontroller (AVR Atmel128 and STK300), a pair of motor driving circuits, and a PC that allows measurements of both normal and shear forces, Figure 4. The normal forces are the result of gripping objects and can be calculated from the mechanical pressure that builds up upon compressing the elastomer material. Shear forces are the result of accidentally missing the object, which slips between the fingers of the artificial hand. Subsequently, the sensor is subject to shear forces, causing if the value is below a limit, activation of the gripping function.

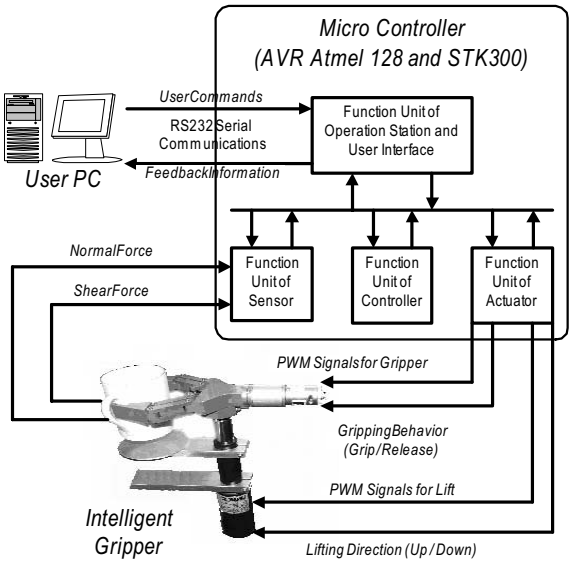

Figure 4. Gripper system setup.

The gripping and lifting motions of the system are controlled by the operator through a RS232 serial communication, and actuated by means of 2 different motors. One is used for the vertical movement (lifting up/down) and the other is for the gripping behavior (opening/closing).
Some of the requirements for the system are:

- A non-slip lift is done by applying enough and necessary normal force (lowest maximum value).
- The applied normal force should behave linearly and cover the weight in the range of 0 to 500 gram.
- The applied normal force should behave linearly the same for cylindrical objects with the diameter between 0 and 10 cm.
- The system could be programmed to place down an object "soft" on a surface.
- It could be possible to build in "intelligence" in terms of characterizing different shapes of objects (cubic, convex, cylindrical or concave).

The design will be presented in a top down specification approach. The oriented signal flow graph, whose arcs are labelled with the signals exchanged between various *FUs* is presented in Figure 5.

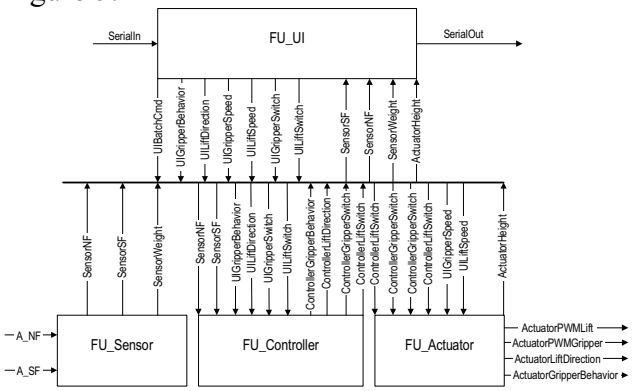

Figure 5. System *FU* Diagram.

There are four virtual nodes (*FUs*) inside the system (mapped on one physical node):

- **FU_UI** (user interface) – status/commands with PC,
- **FU_Sensor** - sampling the forces,
- **FU_Controller** – computing system behaviour, and
- **FU_Actuator** – applying PWM (Pulse-Width Modulation) signals to the motor.

As presented in Section 2, a COMDES system can be seen as a signal flow diagram. It consists of a set of function units, a set of input/output signals, and the mappings among the signals. The specification of the signals exchanged among *FUs* is listed in the Table 1:

Table 1. *FUs* and the Mapping of Input/Output Signals

Function Units	Signals Received	Signals Sent
FU_UI	**SerialIn** (Physical Input)	**SerialOut** (Physical Output)
	SensorNF	UIBatchCmd
	SensorSF	UIGripperSwitch
	SensorWeight	UILiftSwitch
	ActuatorHeight	UIGripperBehavior
		UILiftDirection
		OSLiftSpeed
		OSGripperSpeed
FU_Sensor	**A_NF** (Physical Input)	SensorNF
	A_SF (Physical Input)	SensorSF
		SensorWeight
FU_Controller	SensorNF	ControllerLiftSwitch
	SensorSF	ControllerGripperSwitch
	UIGripperBehavior	ControllerLiftDirection
	UILiftDirection	ControllerGripperBehavior
	UIGripperSwitch	
	UILiftSwitch	
FU_Actuator	ControllerLiftSwitch	ActuatorPWMLift
	ControllerGripperSwitch	ActuatorPWMGripper
	ControllerLiftDirection	ActuatorLiftDirection
	ControllerGripperBehavior	ActuatorGripperBehavior
	UILiftSpeed	ActuatorHeight
	UIGripperSpeed	

3.1 Function Unit of Sensor (FU_Sensor)

The job of FU_Sensor is to receive physical signals generated by the sensors (interrupt routines), ADC, and transmit the values to FU_Controller. There are two physical values to be measured. As a result, two activities, which are in charge of handling the two physical signals respectively, are aggregated into this *FU*. The activities, Act_NF and Act_SF, work as two tasks, handling the incoming sampled values of normal force and shear force. They are running in two different periods. Because shear force is more critical to the safety property of the system, the sampling rate of Act_SF is higher than that of Act_NF.

A series of executions is carried on in each activity. These executions are organized in *CFB*: CFB_NF, which aggregates the execution sequence for calculating normal force; and CFB_SF, which is for the calculation of shear force. The *CFBs* are containers of some basic executions, such as unit conversion and limit checking. The hierarchical structure of FU_Sensor is presented in Figure 6.

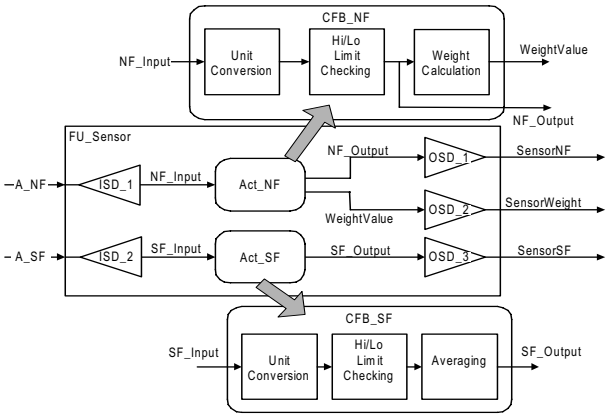

Figure 6. Structure of FU_Sensor and its internal activities.

Every activity has an input signal driver and an output signal driver. The pair of drivers is in charge of decoding or encoding of the transferred signals (messages). Tables 2 and 3 summarize the detailed signal transferring mappings.

Table 2. Mapping of Input Signal Drivers and Corresponding Internal Signals Produced

ISD's	Input Signals	Internal Signals Produced
ISD_1	A_NF (Physical input)	NF_Input
ISD_2	A_SF (Physical input)	SF_Input

Table 3. Mapping of Internal Signals Produced and Consumed Inside the Activity

Activities	Internal Signals Consumed	Internal Signals Procuded
Act_NF	NF_Input	NF_Output
		WeightValue
Act_SF	SF_Input	SF_Output

Table 4. Mapping of Internal Signals and Corresponding Output Signals

OSD's	Internal Signals Consumed	Output Signals
OSD_1	NF_Output	SensorNF
OSD_2	WeightValue	SensorWeight
OSD_3	SF_Output	SensorSF

Subsequently and accordingly will be specified the other *FUs*.

Next level of specification is event-based behaviour, i.e *SMs*. The execution of the control software is time-triggered, i.e. the only external event is the elapse of time period. We will exemplify with the Act_MotorControl activity, from FU_Controller, which controls the power (ON/OFF) and the behaviors (Up/Down and Grip/Release) of the two motors, Figure 7 and Table 5

Guard Condition	Notation
NFValue > NFmin	x_1
SFValue < SlipThrd	x_2
ControllerEnable == true	x_3

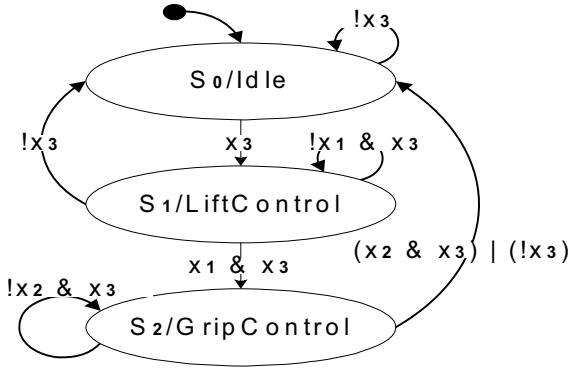

Figure 7. Act_MotorControl *SM* behavior.

Table 5. Next State Mapping

Current State	Trigger	Guard	Next State	Action
s_0	↑(kT)	x_3	s_1	Lift Control
	↑(kT)	$!x_3$	s_0	Idle
s_1	↑(kT)	$x_1 \& x_3$	s_2	Grip Control
	↑(kT)	$!x_1 \& x_3$	s_1	Lift Control
	↑(kT)	$!x_3$	s_0	Idle
s_2	↑(kT)	$!x_2 \& x_3$	s_2	Grip Control
	↑(kT)	$(!x_1 \& x_3) \mid !x_3$	s_0	Idle

4. Verification in COMDES

Analysis of system behaviour is a highly complex problem, which has been extensively studied over the years but no comprehensive solution has emerged as yet [3,5,9]. A practical approach to systems analysis has to take into account the specific features of the system model used. That is the case with component-based design, which is inherently based on the adoption of such a metamodel (framework) that has to explicitly specify all aspects of system structure and behaviour (e.g. COMDES). Compositional specification of component-based systems is naturally combined with compositional verification, which might also be denoted as modular reasoning [7]. In that case the system is partitioned a-priori into subsystems and components with clearly defined functionality and interfaces (e.g. *FUs*, activities, etc.), which makes it possible to verify them separately, and then by using a suitable inference rule, make conclusions about the correctness of the system as a whole.

The COMDES model combines in a natural way reactive (event-related) and transformational (data-related) aspects of system behaviour. At the same time, these two aspects are clearly separated i.e. the model specifies which reactions have to be generated in response to certain events, and how these reactions are to be generated within the corresponding system states. That is why it is possible to treat them in separation: event-driven behaviour can be checked with respect to events, reactions, and discrete abstractions using an appropriate technique, e.g. model checking. Data transformations might be verified via symbolic data manipulations and assertion compositional proofs integrated in model checking through provable assertions [3,9]. Each *FB* or *CFB* have an associated set of provable assertions corresponding to the continuous state trajectory evolving in R^n. Predicates include threshold post condition boolean events and abstracted signals. System correctness properties can be expressed in LTL formulas and the whole model can be translated native in SPIN/PROMELA [3].
For example, if we want to specify a property for the gripper system, we will express it as a LTL formula and then convert it to the so-called never claims.

The lift will not move before the object is well-gripped

The LTL formula, according to some property pattern decomposition, which decodes an

English sentence into temporal/logical connectors, logical terms, etc, will become:

```
<>Well_Gripped•( ! Lift_Up U
                 Well_Gripped)
```

Timing correctness is addressed by applying schedulability analysis with respect to real-time deadlines imposed on the execution of *FU* activities (tasks) and distributed transactions. However, real-time scheduling theory assumes a periodic execution framework, i.e. all activities have to behave as periodic tasks. This poses no problems with activities that are inherently periodic but aperiodic activities have to be modelled as pseudo-periodic (sporadic) tasks. Alternatively, event-driven *SM* might be transformed into equivalent time-driven *SM* having periodic execution patterns [8].

5. Conclusion

The paper has presented the modelling technique of COMDES framework as a possible solution for embedded software development toward mechatronic systems. The main idea behind this development is to provide a metamodel that facilitates the implementation of open yet dependable systems: the system is built from reusable software components that are error-free by design. Furthermore, components interact via softwiring protocols providing for safe and predictable behaviour in an open system environment. System behaviour is specified in terms of concurrent processes (activities) encapsulated in various subsystems, as well process interactions within and across subsystems. Individual process behaviour can be specified in terms of hierarchical and hybrid *SMs*. Behavioural verification is enhanced by the compositional structure of separated functionality, i.e. discrete and continuous, using a combination of symbolic data manipulation integrated in state transition model checking. This is a powerful metamodel, applicable to a broad variety of control systems, such as continuous and sequential controllers, etc.

Given the complexity of the mechatronic systems, we have advocated a framework that would ultimately result in a feasible embedded software development method.

References

[1] C. Angelov, K. Sierszecki and N. Marian (2005) *Design Models for Reusable and Reconfigurable State Machines*, LNCS 3824, pp. 152 – 163

[2] C. Angelov and K. Sierszecki (2004) *A Software Framework for Component-Based Embedded Applications*, Proc. of the Asia-Pacific Software Engineering Conference APSEC'2004, Busan, Korea

[3] N. Marian, Yang Liu and Yue Lu (2005) *Improvements in Control Software Development*, Proc. of the 6th International Workshop on Research and Education in Mechatronics REM'2005, Annecy, France

[4] N. Marian, Jinpeng Ma (2005) *Reliable Component-based Software Development for Embedded Systems*, Proc. of the 2nd International Conference on Mechatronics ICOM'05, Kuala Lumpur, Malaysia

[5] Wagner F., Wolstenholme P. (2003) *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

[6] Wang S., Shin K. G. (2000) *An Architecture for Embedded Software Integration Using Reusable Components Proc. of International Conference on Compilers*, Architecture, and Synthesis for Embedded Systems, San Jose, California

[7] B. Berard, M. Bidoit et al. (2001) *Systems and Software Verification*. Model-Checking Techniques and Tools, Springer

[8] A. Benveniste, P. Caspi et al. (2003) *The Synchronous Languages 12 Years Later*, Proc. of the IEEE, vol. 91, No 1, pp. 64-83

[9] Nanette Bauer, Ralf Huuck (2001) *Towards Automatic Verification of Embedded Control Software*, Proc of the IEEE Asian Pacific Conference on Quality Software, Hong Kong