VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

CODE GENERATION FOR SEMANTIC EVOLUTION OF EMBEDDED SYSTEMS

Smt. J. Sasi Bhanu[1], A. Vinaya Babu[2] and P. Trimurthy[3]
[1]Department of Computer Science Engineering, KL University, Vaddeswaram, Guntur District, India
[2]Department of Computer Science and Engineering, JNTU Hyderabad, India
[3]Department of Computer Science and Engineering, ANU Guntur, India
E-Mail: sasibhanu@kluniversity.in

ABSTRACT

Monitoring and controlling a safety or a mission critical system (Production system) is normally undertaken through interfacing with an embedded system (Target System). The embedded system which is meant for monitoring and controlling of either mission critical or safety critical system cannot be shut-down for want of making changes to ES software either to fix existing bugs, enhance performance, add fault tolerance or add more functionality due to the reasons that it is quite expensive to shut-down and restart either the production system or the Target system. Changes to the ES software, thus have to be carried while the ES system is up and running. The process of making changes while the ES system is up and running is called dynamic semantic evolution. Many methods have been proposed in the literature using which the dynamic semantic evolution can be carried. One of the methods is related to generation of code, which is stored starting from a memory location and creation of a task out of the code stored and make it to run waiting for an event to take place. In literature it has been mentioned that dynamic semantic evolution can be undertaken through a module generation process but the implementation of the same has not been presented. In this paper, an algorithm and the process of implementation of the same has been presented. The same has been experimented on a prototype application that monitors and controls temperatures within a nuclear reactor system.

Keywords: dynamic semantic evolution, embedded system, code generation, safety & mission critical systems, ES architectures.

1. INTRODUCTION

A Target system which is interfaced with a production system cannot be shut-down for want of making changes to the embedded software as bringing down and bringing up the production system is a very expensive proposition and sometimes such types of systems cannot be shutdown at all. The software changes to the ES software are required to fix an existing bug, increase the performance and implement fault tolerant systems. Some times more of the control functions are to be added. Thus the ES software must evolve dynamically as the ES system is up and running.

Many architectures have been presented in the literature using which the dynamic semantic evolution can be carried. Each of the architecture presented in the literature dealt with a different method with which dynamic semantic evolution of loaded systems can be carried. Very few architectural methods have been presented that are related to embedded systems. Each architecture presented in the literature deal with only one type of semantic evolution. It is necessary to consider all the evolution methods and provide a unified architecture and the kind of dynamic evolution to be adapted should be decided at run-time. The comprehensive architecture that caters for all types of dynamic evolutions of embedded systems is shown in the Figure-1.

Dynamic semantic evolution of the embedded system can be achieved through Data Approach (Invoking and deleting the existing Tasks as per the desired functionality), Rule based Approach (Invoking the task execution as per the rules for which a repository can be maintained), Module generation and invoking approach (Generating new modules based on HOST sent specification), Module copy approach (Creation of the new modules as per the code sent from the remote HOST), Module update approach (Creation of update module and swapping the old module with the Update module), Module update with attached criticality assessment Approach (Create update modules and subject the module for criticality assessment).

The overall architecture shown in Figure-1 includes all the above mentioned methods and therefore can be considered as comprehensive and efficient. The type of evolution that must be taken up can be dictated from the HOST through transmission of appropriate commands. Semantic Evolution block can implement the kind of semantic evolution that needs to be implemented as per the request initiated from the HOST.

In literature many architectural models have been presented narrating all the theoretical foundations which are used in building those architectures, but the way those architectures have been implemented have not been presented except for the implementation of the dynamic evolution of a single JAVA, C++ or C program. No architecture as such has been presented that is suitable for dynamic semantic evolution of ES software. In this paper an algorithmic approach and the process used for implementing semantic evolution of ES software using module / Code generation approach has been presented.

2. PROBLEM DEFINITION

Many methods have been proposed in the literature for implementing dynamic semantic evolution of loaded systems and some methods related to semantic evolution of embedded systems, the implementation of which however have not be presented except dynamic evolution of a single program which are written in JAVA, C++ and C. The components included into architectural presentations needs to be implemented to prove the concepts that have been described for achieving dynamic semantic evolution of embedded systems.

Many dynamic semantic evolution methods have been proposed in literature which includes among many, the method that generates the code online and make the code converted as a task which is scheduled and make the task work in defined real time environment. In this paper the implementation process, algorithm has been presented. The specification related to code generation is transmitted from the HOST and the specification is used for generating code, copying the same to the designated area in the memory, creating a Task under an RTOS and scheduling the task waiting for an event to take place. The algorithm has been applied to generate code required to construct ES software related to a system that monitors and controls the temperatures within a nuclear reactor system.

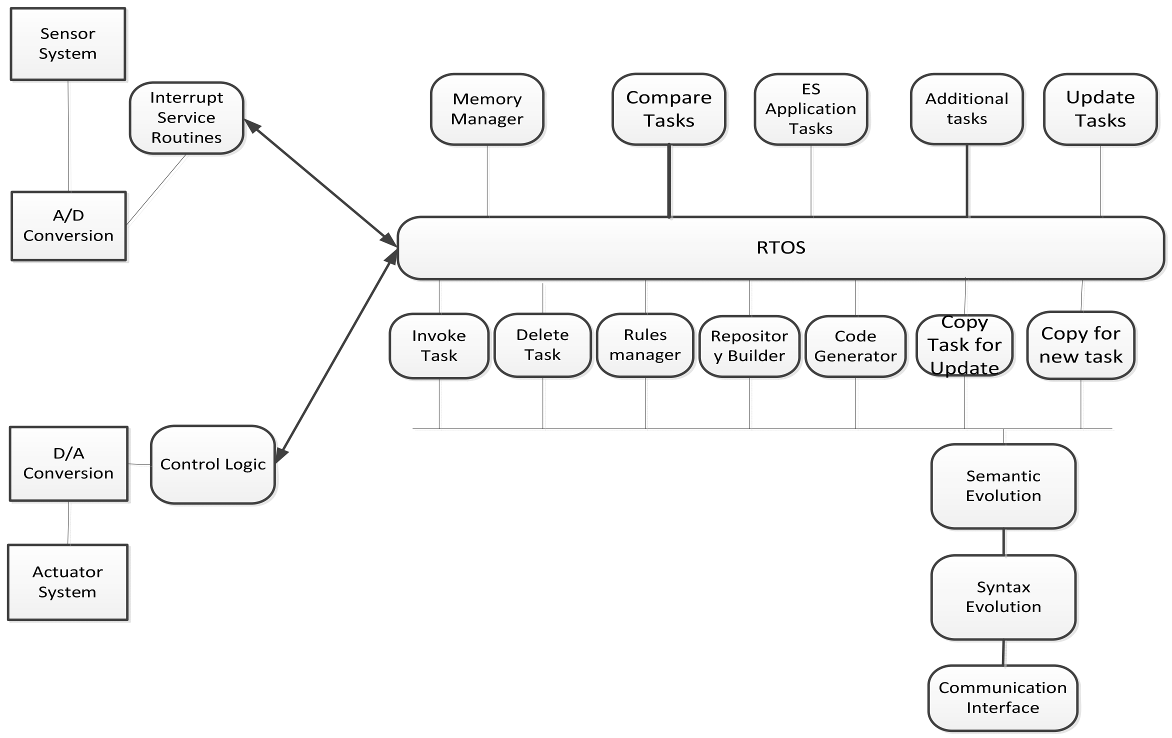

Figure-1. Overall semantic evolution architecture.

3. LITERATURE SURVEY

Two code generation frameworks are necessary for generating the code in respect of Embedded Hardware and software. Code is required in respect of the Hardware to facilitate selection, integration and displaying the layout and also code generation is required in respect of Embedded Software. Many frameworks have been presented in the past to generate code relating to presentation of foot print. Sastry *et al.*, [1] have presented a method to generate the code meant for modelling the Hardware for facilitating the selection, integration and displaying the layout of the Hardware interfacing.

Clean room software engineering (CRSE) methodology included the use of standard structures like if-then-else but no formal framework has been presented for generation of the code. Code is required for two purposes. Code is required for designing hardware that includes selection, integration and display. The internal functioning of each of the hardware component can also be represented in terms of the code.

When it comes to ES software, CRSE advocated that code be generated using state model. The state model includes both Hardware and Software states. When a system is in a hardware state other than

VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.
www.arpnjournals.com

the Micro Controller state, no code is executed. Some electronic function is being processed in that case. Most of the code is executed when the embedded system is in software state at which time code is executed within the microcontroller. The hardware states are not that important for generating the code related to embedded software.

Dionision de Niz *et al.*, [2] have presented a model based code generation framework for embedded real time system considering multiple operating systems, communication mechanisms, different hardware sizes and dynamic structures of the software. A number of design and code generation approaches have been presented by Gang Zhou *et al.*, [3] for development of embedded software that require implementation of concurrency. W. Thies *et al.*, [4] have presented Framework called Stream IT which is based on data flow formalism. Real time workshop (RTW) from math work [5] considered code generation based control system modelling and time has been considered as integral part of the model development. E. Kohler *et al.*, [6] N. D. Jones *et al.*, [7] have proposed partial evolution methods that help automating the designing and code generation process.

Chen Xi Lu *et al.*, [8] have used partial evaluation methods for optimized code generation by transforming a generalized actor based model to a target code while preserving the models semantics Gang Zhou *et al.,* [3] have generated C Code based on the Models. The issues related to co-existence of hardware and software has not been considered. Chen Xi Lu *et al.*, [8] have proposed a method of generating SystemCTLM Model from UML specification. Using the SustemCTLM Model, System CRTL Model embedded software is generated. The System CRTL model is synthesized to generate FPGA RTL code in the VHDL language and a ASIC netlist is generated. Chen Xi Lu *et al.*, [9] have proposed a method to generate embedded software from state diagrams. Prior to the generation of the code the state charts are validated for correctness related to unused states, one initial state, states with outgoing links, reachability of every state from initial state, existence of at least one final state, availability of at-least one path that reaches the final state even if a loop is inexistence at any of the state.

Matteo Bordin *et al.*, [10] have investigated the fitness criteria for a programming language that can be used to generate the Embedded Software from model based specifications. They have checked the fitness of Java language from the point of object oriented semantics and the ability to support the issues related to concurrency. Marko Hannikainen, Jarno Knuutila *et al.*, [11] have used SDL (Specification Description Language) for modelling complex real-time embedded system. SDL is a Graphic and formal language. User can interact with SDL and build embedded software from the scratch. Kjeld H. Mortensen [12] has proposed building a system not necessarily the embedded software using the Coloured Petri net models (CPN). The model is debugged and verified for its exactness suiting to the application.

Kathy Dang Nguyen, Zhenxin Sun [13] have explained that an embedded system can be developed in terms of UML Artefacts. They have proposed that System C code can be generated by way of establishing the System C equivalents to UML notations. They have added some standard extensions to UML so that the UML notations can be mapped to System constructs. Luis Gomes and Aniko Costa *et al.*, [14] have recommended a method to generate code given a set of hierarchical state Charts. They have also explained the way the hierarchical or concurrent processing state charts can be developed.

Code generation methods proposed in the literature have not considered the perspective of dynamic semantic evolution of the embedded software. The state models however can be effectively used for generation of ES software through use of Dynamic semantic evolution process.

4. INVESTIGATIONS AND FINDINGS

Runtime module generation is a very powerful technique that allows new modules to be generated at run time to enable the system to be adapted at run time. Generation of new modules is a complex process. Code generation is a time taking process and also it is difficult to generate efficient code based on the inputs fed from a remote HOST about the changed situation. Huge repository has to be built within the embedded system which is required for generating the code

Figure-2 shows the architecture for runtime Module generation technique. The details required for code generation are transmitted from the HOST and the same is maintained within the embedded system. Two processes are to be included into the architecture one for maintaining code generation repository and the other for generating code. The memory module will make available the address space required for storing the code that is generated. After creating the code a task is created along with the event for which the task must be waiting. The tasks that are related to repository building, code generation and memory management are invoked by the semantic evolution modules triggering their related events.

Repository building

Repository building requires that data is obtained from the HOST and different Tables are maintained. Table-1 to Table-9 shows the repository required for code generation. The data about these tables is sent using command and the tables or maintained by the repository builder. The required arguments to refer to the kind of table into which the data has to be entered is also sent along with the command.

Algorithm for generating the code

The freely available memory pools are obtained by making a function call to the RTOS and the memory pools are used to write the code generated. The following algorithm is run and the code is generated and the same is written to the memory area. The algorithm is a 8 step algorithm which is detailed below

Step 1: Construct a library of the code segments

A code segment is a unit of execution that is required quite frequently. The code units that are related to communicating with the hardware devices and the code units that are used in non-member functions are recognized and identified as the code segments. Every code segment will clearly recognize variables, their types and location where the variables must have been declared including the details of locations which include, Global, instance, local, functional arguments and return variables. This way of categorizing the variables will clearly help in mapping and constructing code functions and the class modules in which the code segments are placed.

The code segments are pre-identified and developed as a library over a period of time. The library is maintained and more number of code segments is added to the library as when needed especially when a ES application is analysed and designed. A code segment may include many other code segments or may even call code functions.

A repository of code segments can be maintained which will be used latter for resolving and placing the variables in appropriate locations. The details of the repository constructed for locating and maintaining the standard code structures that help generating the code for TMCNRS is shown in the Table-1. Example standard code structure for writing data to an LCD is shown below:

```
# LCD Command write segment (LCD-COMMAND-
  WRITE)
# Instance Variables
# Local Variables
# Global Variables
# Functional Variables
Int d;

# includes busycheck ()
# Code Begin
Rs = reset;
P2=comm.;
en =set;
# includes delay (d);
en =reset;
# includes delay (d)
# includes busycheck ();
# Code End
```

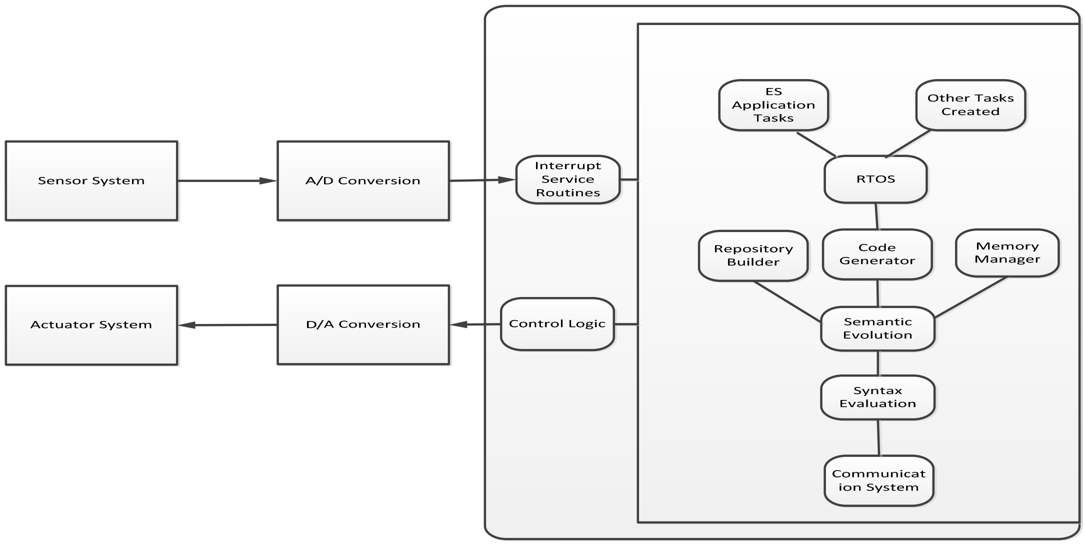

Figure-2. Semantic evolution based on code generation.

Step 2: Construct a library of non-member functions

Code functions may be member functions or non-member functions. The functions that are mapped to the classes shall be the member functions and the remaining functions shall be non-member functions but shall be declared as friend functions with the classes wherever needed. When non-member functions are mapped to the classes, the same will be defined as friend functions.

A library of standard non-member functions are identified and maintained as shown in the Table-2. The Mapping of the code segments to the Non-Member functions is also shown in the Table-2. When code segments are included into a function, the variable declarations as instance, global local, and return or functional shall be automatically get projected and maintained.

Step 3: Construct a library of hardware related code functions

Some of the code functions are member functions of the class's that are related to hardware devices. A library of standard member functions that are related to hardware are identified and maintained as shown in the Table-3. The mapping of the code segments to the member functions of the hardware classes is also shown in the Table-3. When code segments are included into a function, the variable declarations as instance, global local, and return or functional shall automatically get projected and maintained.

Step 4: Map the non-member functions to the classes

The non-member functions are mapped to various classes as friend functions as necessary. The mapping of the non-member functions to the classes has been shown in the Table-4. When non-member functions are mapped to the classes, the global variables and instant variables will be located at global declarations or instance declarations as the case may be. When functions are mapped, the variable declarations shown in the function definition will be automatically located either as Global or instance variables

Step 5: Map the hardware dependent functions to the classes that are related to the hardware

The hardware dependent functions are mapped to the classes that implement the interface to the hardware devices. The data repository showing the mapping is shown in the Table-5. When functions are mapped, the variable declarations shown in the function definition will automatically get located either as global or instance variables.

Step 6: Map the hardware dependent variables to the classes that are related to the hardware

Some of the classes that are related to the hardware just have variables that are defined to map to the PINS and ports of the Micro controller to which the hardware devices are connected. Signals are asserted on the PINS of the micro controller by way of setting the memory variables with appropriate values. Data variables which are mapped to the PINS are inserted into the classes based on the connectivity details. The mapping of hardware dependent variables to the classes are shown in the Table-6.

Step 7: Maps the support oriented functions to the supporting classes

Some of the functions are supporting functions that are necessary for undertaking specialized activities. Such functions are mapped to the classes that provide supporting services to other classes. The mapping of supporting functions to the supporting classes is shown at the Table-7. When supporting functions are mapped, the variable declarations shown in the function definition automatically get located either as global or instance variables.

Step 8: Maps Task oriented functions to the classes that are self-looped

Some of the functions are Task oriented functions that are necessary for undertaking Task oriented activities. Such functions are mapped to the classes that provide task execution services to other classes. The mapping of task oriented functions to the task oriented classes is shown at the Table-8. When task oriented functions are mapped, the variable declarations shown in the function definition will be automatically located either as global or instance variables.

Step 9: Maps the class functions as the entry procedures of system states

A system enters into a state due to transition from the previous state as a consequence of occurrence of an event while the system is in previous state. When a system enters into a state or exiting from a state, some procedures are executed. Some procedures can also be executed while the system is within the state. In the previous steps, a process which flushes the classes with the methods and attributes and also that establishes the global behaviour of the embedded system has been explained. The mapping of the class methods to entry procedure of system states will define the state boxes comprehensively. The system is built using several objects and each of the objects be in different states and execute procedures when the system enters into a particular state. Table-9 shows various objects and states into which the objects undergo transitions and the kind of procedures executed for realizing a user requirement.

Step 10: Generate main control of execution

The following procedure helps building the main function in which central control logic is situated.

a) Capture state transition details as per the details shown at Table-9
b) The procedures that are mapped to the superstates which are self-looping states shall be recognized as

the task that must be scheduled within the real time operating system.

c) Create a buffer of such schedulable tasks. For each of the task create a stack as a character array of a default length of 3000 bytes
d) For each of the non-schedulable super-state, find sub-state from the Table-9.
e) Find all the state flows from table-9 by following Right to left and top to bottom rule
f) For each of the sub-state flow find all the entry procedures and include the in the main function.
g) If any of the sub-state is self- looping in nature, include infinite while loop and enclose the code within the scope of the while loop or create a finite looping through a for-statement or while-statement using the finiteness as the controlling parameter.
h) If a class is associated with the function/procedure to be executed as a part of entry procedure for the first time, an instance of the class is included before the method is called
i) Include code to start the operating system
j) Consider each of the self-looping super states and for each of the self-looping super state, create objects that are related to the entry procedures of the self-looping superstate
k) Include an RTOS statement for creating a task representing the entry procedure of self-looping super state, under the control of RTOS including the reference to stack buffer that is created earlier
l) Include statements to start RTOS

Step 11: Generate total code

The generation of total code thus involves the following steps

a) Construct library of standard code segments duly identifying various types of variables
b) Construct library of the standard non-m ember functions through code segments and while doing so, consolidate types of variables especially the global and instance variable. The local, return and functional variables are absorbed with in the functions
c) Construct library of the standard hardware related functions through code segments and while doing so consolidate types of variables especially the global and instance variable. The local, return and functional variables are absorbed within the functions
d) Create hardware dependent variable in the classes related to hardware devices duly mapping the PINS of micro controller to the memory address variables
e) Construct library of the standard Member functions through code segments and while doing so consolidate types of variables especially the global and instance variable. The local, return and functional variables are absorbed within the functions
f) Map the Non-member functions to the respective classes
g) Map hardware dependent functions to hardware processing classes
h) Map support functions to the support classes
i) Map the task oriented function to task oriented classes
j) Include into entry procedures of each state the functions related to various types of classes.
k) Build a main method by tracing the super state sub state entries in table-9.

The semantic evolution of ES software is achieved through tasks which include repository building task and code generation task

Repository builder task

This task will be waiting for an event triggered by the Semantic Evolution Task. As and when the event is triggered the task commences it execution. This task maintains several look up tables the data for which is obtained from the remote host. The tables are temporarily maintained and the data in the tables is deleted after the code is generated out of the specification stored in the lookup tables. After the lookup tables are created, the task assigns itself to the event for which it has been waiting.

Code generation task

This task will be waiting for an event triggered by the Semantic Evolution Task. As and when the event is triggered the task commences it execution. This task Obtains a free memory pool by calling a RTOS function which returns the start address at which the Task can be created. Code is generated using the lookup tables shown in Tables 1 to 9, and the code is stored starting from the memory location returned by the RTOS. An entry is made into the Task lookup table which is also stored with the name of the event with which the task should be triggered. Memory lookup table is also updated with the pool identified with the task to which the memory is assigned. A task is created under the RTOS so that same is invoked and will be waiting for its related event to occur.

Table-1. Repository of standard code structures.

Serial number	Coding structure name	Local variables		Global variables		Instance variables		Argument variables		Return variables	
		Name	Type	Name	Type	Name	Type	Name	Type	Name	Type
1	LCD-BUSSY-CHK()	busy	int	Set =01	int						
				Reset=00	int						
				rs=P2^5	sbit						
				rw=P2^4	sbit						
				en=P2^3	sbit						
				busy=P2^7	sbit						
2	LCD-COMMAND-WRITE()							d	int		
3	LCD-DATA-WRITE()			Set =01	int						
				Reset=00	int						
				rs=P2^5	sbit						
				rw=P2^4	sbit						
				en=P2^3	sbit						
				busy=P2^7	sbit						
4	LCD-NDATA-WRITE()			Set=01	int			Data[20]	char		
				Reset=00	int						
				rs=P2^5	sbit						
				rw=P2^4	sbit						
				en=P2^3	sbit						
				busy=P2^7	sbit						
5	RS232C-RECV()	Include.h						dat	Uchar		
6	RS232C-SEND()							dat	Uchar		
7	I2C-READ()			SDAt=P1^5	sbit			i	Ucar		
				SCLt=P1^4	sbit			dat	char		
8	PROCESS-DELAY()							i	long		

Table-2. Repository of standard non-member functions.

Serial number	Standard code function	Coding structure name	Local variables		Global variables		Instance variables		Argument variables		Return variables	
			Name	Type	Name	Type	Name	Type	Name	Type	Name	Type
1	I2C_read-Temp()	I2C-READ	i	Char	SDAt=P1^5	sbit			i	Ucar		
			dat	char	SCLt=P1^4	sbit			dat	char		
2	Dealy()	PROCESS-DELAY							i	long		
3	I2C_write_Temp()	I2C-WRITE	i	Char	SDAt=P1^5	sbit			i	uchar		
			dat	char	SCLt=P1^4	sbit			dat	char		
4	I2c_start-temp()	I2C-START			SDAt=P1^5	sbit						
					SCLt=P1^4	sbit						
					HIGH=1	Int						
					LOW=1	int						
5	I2c_stop_Temp()	I2C-STOP			SDAt=P1^5	sbit						
					SCLt=P1^4	sbit						
					HIGH=1	int						
					LOW=1	int						
6	asciiToHex()	ASCII-TO-HEX			Digit0	Char			value	char	dat	unsigned
					Digit1	Char						
					Ascii0	Char						
					Ascii1	Char						
					Ref1	Int						
					Ref2	int						
7	HexToAscii()	HEX-TO-ASCII	temp	usgined	Digit0	Char			value	char		
					Digit1	Char						
					value	char					dat	usigned

VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

Table-3. Repository of standard hardware related member-functions.

Serial number	Standard function name	Coding structure name/ function name	Local variables		Global variables		Instance variables		Argument variables		Return variables	
			Name	Type	Name	Type	Name	Type	Name	Type	Name	Type
1	readkey()	I2C_strat-temp()										
		I2c_write_Temp(0x70)										
		I2c_write_Temp(0xFE)										
		I2c_stop_Temp()										
		I2c_start_temp										
		I2c_write_temp(0x71)										
		I2c_read-temp()										
2	write_Command()	LCD-COMMAND-WRITE			Set =01	int			d	int		
					Reset=00	int						
					rs=P2^5	sbit						
					rw=P2^4	sbit						
					en=P2^3	sbit						
					busy=P2^7	sbit						
3	Busy()	LCD-DATA-WRITE			Set=01	int						
					Reset=00	int						
					rs=P2^5	sbit						
					rw=P2^4	sbit						
					en=P2^3	sbit						
					busy=P2^7	sbit						
4	Recv()	RS232C-RECV	Include.h						dat	Uchar		
5	Data_write()	LCD-DATA-WRITE			Set =01	int						
					Reset=00	int						
					rs=P2^5	sbit						
					rw=P2^4	sbit						
					en=P2^3	sbit						
					busy=P2^7	sbit						
6	Send()	RS232C-SEND							dat	Uchar		

VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

Table-4. Mapping non-member functions to the classes.

Serial number	Name of the class	Delay()	I2c-read	I2c-write	I2c-start	I2c-stop	Hex-Ascii	ASCII-HEX	Get-Reference
1	Initialization process	√					√		
2	Process key		√	√	√	√		√	
3	Temp1Task	√	√	√	√	√	√		√
4	CompareTemp1Task						√		
5	Temp2Task	√	√	√	√	√	√		√
6	CompareTemp2Task						√		
7	ProcessTemp1Temp2 Task	√					√		

Table-5. Mapping member functions to the hardware classes.

Serial number	Name of the class	Readkey ()	Command -write()	Data-write()	Busycheck ()	Send ()	Receive ()
1	Process Key	√					
2	Process LCD		√	√	√		
3	Process HOST					√	√

Table-6. Mapping supporting functions to the supporting classes.

Serial number	Name of the class	Compare Passwd	CompareTemp1withRef	CompareTemp1withRef
1	Validate Passwd	√		
2	CompareTask1		√	
3	CompareTeask2			√

Table-7. Mapping task functions to execution task classes.

Serial number	Name of the class	Display Init Message	Display enter passwd message	Read Ref Temp	Convert reference digits	Temp1Processing Task	Temp2 process Task	compare Temp1 Temp2 Task
1	Initialisation process	√	√	√	√			
2	Temp1Task					√		
3	Temp2Task						√	
4	ProcessTemp1Temp2Task							√

Table-8. Function mapping to the classes.

Name of the sequence	**Name of the object**	**Type of the object**	**Name of the state**	**Entry procedures**
Display InitMessages based on the reset button	**Operator**	**Human**	**Operator**	-
	Micro Controller	Hardware	Micro Controller	-
	Initialization process	Software	InitMessages	InitialisationProcess.initMessages() { ProcessLCD.Command_Write()ProcessLCD . Data_write() }
				InitialisationProcess. displaypasswdMessaage() { ProcessLCD.Command_Write()ProcessLCD . Data_write()}
	Process LCD	Software	LCDProcess	ProcessLCD.Command_Write()ProcessLCD . Data_write()
	LCD	Hardware	LCD	-
Read password through keystrokes	Name of the object	Type of the Object	Name of the State	Entry Procedures
	KeyBoard	Hardware	Keyboard	-
	ATOD Converter	Hardware	ATOD Converter	
	Micro Controller	Hardware	Micro Controller	-
	Initialisation Process	Software	Process Key	Process Key. Readkey ()
	Process LCD	Software	LCD Process	Process LCD. command_write () ProcessLCD. data_write ()
	LCD	Hardware	LCD	-

VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2015 Asian Research Publishing Network (ARPN). All rights reserved.

Table-9. State transition details.

Serial number of the state	From state	To state	Whether self-looping	Whether starting state
1.	Keyboard	ATOD Converter		
2.	ATOD Converter	Micro Controller		
3.	Reset	Micro Controller		Yes
4.	Micro Controller	Initialization and Main Control		
5.	Micro Controller	Init Messages		
6.	Micro Controller	Process Key		
7.	Init Messages	Process Key	Yes	
8.	Process Key	Validate Password		
9.	Validate Password	Init Reference		
10.	Init Reference	Init Reference-1		
11.	Init Reference-1	Init Reference-2		
12.	Init Reference-1	Process Host		
13.	Process Host	Micro Controller		
14.	Init Messages	LCD Process		
15.	Process Key	LCD Process		
16.	Validate password	LCD Process		
17.	Init Reference	LCD Process		
18.	Reference-1	LCD Process		
19.	Reference-2	LCD Process		
20.	LCD Process	LCD		
21.	Initialization and Main Control	Temp1	YES	
22.	Initialization and Main Control	Temp2	YES	
23.	Temp1	Compare Temp1 with Remp2	YES	
24.	Temp2	Compare Temp1 with Remp2	YES	

5. CONCLUSIONS

The ES software stored within an embedded system which is used for monitoring and controlling the operations within a safety critical or mission critical system must be evolved dynamically while the system is running. Many methods have been proposed for dynamic evolution of the ES software. One of the methods recommended is online code generation based on the code specification sent from the HOST that is predominantly cantered around the object model.

The code generated needs to be located at the memory locations returned by an RTOS and task must be created, schedule under the RTOS and made to wait for the occurrence of an event which completely fits into the event model around which dynamic evolution of embedded software is supported.

REFERENCES

[1] Sastry JKR, V. Chandra Prakash, Bala Krishna Kamesh, S. Venkateswarlu. 2011. Code Generation for Hardware Modeling through Clear Box Structures. International Journal of Communication Engineering Applications-IJCEA. 2(3).

[2] Dionision de Niz, Raj Rajkumar. 2004. Glue Code generation: Closing the Loop Hole in Model Based Development. IEEE Real-Time and Embedded Technology and Application Symposium (RTAS2004).

[3] Gang Zhou, Man-Kit Leung and Edward A. Lee. 2007. A Code Generation Framework for Actor-Oriented models with Partial Evaluation. ICESS2007, Springer-Verlag Berlin Heidelberg. pp. 786-799.

[4] W. Thies, M. Karczmarek and S. Amarasinghe, Stream It. 2002. A Language for Streaming Applications. Proceedings of the 2002 International Conference on Compiler Construction. Springer-Verlag LNCS, Grenoble, France.

[5] http://www.mathworks.com/product/simulink.

[6] E. Kohler, R. Morris and B. Chen. 2002. Programming language optimizations for modular router configurations. Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[7] N. D. Jones, C. K. Gomard, and P. Sestoft. 1993. Partial Evaluation and Automatic Program Generation Prentice-Hall.

[8] Chen Xi Lu Jian Hua Zhou Zu Cheng and Shang Yao Hui. 2005. Modeling System C Design in UML and Automatic Code Generation. ASP-DAC.

[9] Felix Lindlar and Armin Zimmermann. 2008. A codegeneration tool for embedded automotive systems based on finite state machines. IEEE international Conference on Industrial Informatics (INDIN2008), Korea.

[10] Matteo Bordin and Tullio Vardanega. 2007. Real-TimeJava from an automated code generation perspective. JTRES'07, Vienna, Austria.

[11] Marko Hannikainen, Jarno Knuutila, Antti Takko, Timo Hamalainen and Jukka Saarinen. 2000. Automatic C-Code generation from SDL for a wireless MACPROTOCOL. IEEE International Symposium on Intelligent Signal Processing and Communication System (ISPACS 2000) Honolulu. 1: 533-538.

[12] Kjeld H. Mortensen. 1999. Automatic Code Generation from Colored Petri Nets for an Access Control System. Proceedings of 2nd workshop on practical use of Colored Petri nets and Design CPN1999.

[13] [Kathy Dang Nguyen, Zhenxin Sun, P.S. Thiagarajan and Weng -Fai Wong. 2004. Model-driven SoC Designvia Executable UML to System C, Real-Time Systems Symposium. Proceedings 25th IEEE International. 5-8 December. pp. 459-468.

[14] Luis Gomes and Aniko Costa. 2003. From Use cases to system implementation: State chart Based Co-Design. Proceedings of the first ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03).