2009 14th IEEE International Conference on Engineering of Complex Computer Systems

UML Modeling and Formal Verification of control/data driven Embedded Systems

Fateh Boutekkouk and Mohamed Benmohammed
Department of Computer Science
University of Larbi Ben M'hedi
Oum El Bouaghi, Algeria
{Fateh_boutekkouk,ibnm}@yahoo.fr

Abstract—**In this paper, we present our approach for UML based modeling of control/data driven Embedded Systems. In our case application is presented as a network of hierarchic data driven and control driven tasks that communicate via abstract channels. Hardware platform is modeled as UML structure diagram. Mapping of application on hardware platform is modeled through UML constraints. From UML models, a Maude specification is generated. We use this formal specification to formally validate system functionality against some undesirable properties and to estimate system power consumption at a high level of abstraction.**

Keywords-UML; Embedded Systems; Rewriting Logic; Maude; Formal Verification; Power Consumption

I. INTRODUCTION

The ever increasing complexity in both applications and integration density in semi-conductor technology, and strict time to market window have pushed Embedded Systems (ESs) specialists to borrow some technologies and ideas from software engineering. Among these technologies, we find in particular, the Unified Modeling Language (UML) [3] and formal techniques. According to authors, UML can be tailored to different application domains by the definition of profiles. A profile extends an application specific UML sub-set using extension mechanisms offered by UML like stereotypes, constraints, and tagged values. Since UML does not dictate any particular development process to be used, it is on designers to define a design flow. We think that the Y-chart [1] approach is the most appropriate.

On the other side, Maude [5] is a formal language based on rewriting logic [4]. Our choice of this language was not only based on its expression power, but also on its possibility to do simulation and formal verification. In this context, we propose to use UML as a front-end for ESs modeling, high level power consumption estimation and formal verification following the Y-chart principles. The first step in our approach consists in application, architecture, and mapping modeling using UML structure diagrams and constraints. Data flow driven tasks are modeled through UML activity diagrams with coarse grained actions (CGAs) following the Kahn Process Networks (KPN) model of computation semantics.

Control driven tasks are modeled as finite state machines using UML statecharts. Similarly, the hardware architecture is modeled as a network of generic components. Each component is characterized by a set of parameters matching the abstraction level of application. Mapping is modeled as UML constraints. In order to formally validate the application functionality against some undesirable properties like deadlock and perform high level power consumption estimation, we propose to transform the UML models to a Maude specification.

The paper is organized as follows: in Section two we first put the light on rewriting logic and the Maude language. Our approach for ESs modeling using UML is presented in section three. The passage from UML to Maude and properties specification are the subjects of sections four and five respectively. Section six presents an illustrative example and some results. In section seven we overview quickly the related works before concluding.

II. REWRITING LOGIC AND MAUDE

The rewriting logic (RL) was introduced by Meseguer [4]. In RL, the logic formulas are called rewriting rules. They have the following form: **R: [t] -> [t'] if C**. Rule R indicates that term t is transformed into t' if a certain condition C if verified. Term represents a partial state of a global state S of the described system. The modification of the global state S of the system to another state S' is realized by the parallel rewriting of one or more terms that express the partial states.

Maude [5] is a specification language based on the rewriting logic. Two specifications level are defined. The first level concerns the system specification, while the second one carries on the properties specification.

The system specification level is provided by the rewrite theory. For a good modular description, three types of modules are defined in Maude. Functional modules allow defining data types and their functions through equations theory. System modules define the dynamic behavior of a system. This type of modules extends functional modules by introducing rewriting rules. Finally, there are the object-oriented modules that can be reduced to system modules.

Figure 1 illustrates the use of a system module *BANK-ACCOUNT* to define an object counts banking *A* and the two operations capable to affect its content *credit* and *debit* while executing the rewriting rules defined in this module. Note that after the execution of the unconditional rule [credit], the message *credit(A, M)* is consumed and the content of the account is increased. In the same way, the execution of the conditional rule [debit] requires that the condition (N>=M) be verified. The execution of such rule generates the consumption of the message *debit(A,M)* and the reduction of the content of the account.

978-0-7695-3702-3/09 $25.00 © 2009 IEEE
DOI 10.1109/ICECCS.2009.30

IEEE computer society

```
mod BANK-ACCOUNT is
protecting INT .
including CONFIGURATION .
op Account : -> Cid.
op bal :_ : Int -> Attribute .
ops credit debit : Oid Nat -> Msg .
var A : Oid . vars M N :Int
rl [credit]: < A : Account | bal : N > credit(A, M) =>
< A : Account | bal:N + M >
crl [debit] : < A : Account | bal : N > debit(A, M) =>
< A : Account | bal : N - M > If N >= M .
endm
```

Figure 1. The BANK-ACCOUNT module in system module form

The property specification level defines the system properties to be verified. The system is described using a system module. By evaluating the set of states reachable from an initial state, the model-checking allows to verify a given property in a state or a set of states.

The Model-checking supported by Maude's platform essentially uses the LTL logic for its simplicity and the defined decision procedures it offers. After specifying the behavior of its system in Maude system module, the user can specify several predicates expressing some properties related to the system. The user can call the *modelCheck* function while specifying a given initial state and a formula. Maude model-Checker verifies if this formula is valid in this state or the set of all reachable states form the initial state. If the formula is not valid, a counter example (*counterexample*) is displayed. The counter example concerns the state in which the formula is not valid.

III. UML BASED MODELING

The application model is a network of concurrent tasks communicating via abstract channels.

To support hierarchy, we introduce a stereotype called "*Module*". This stereotype is applied on UML composite objects and designates a hierarchic task. Using UML, each task is modeled as an active object (class instance) stereotyped by either *"Dtask"* for flow data driven tasks or "*Ctask*" for control driven tasks. Each task object has one or more than one port. Each port is characterized by its name and the size of the transmitted data in term of tokens number. We denote a token an abstract data item.

The policy of abstracting data is another powerful principle in our approach. Doing this helps in performing fast simulations and formal verifications. Channels are characterized by a point-to-point communication between two tasks. These tasks can be running on the same hardware resource as well as on different hardware resources. In our case, we distinguish between two types of channels: data channels carrying data items and control channels transporting control data. As mentioned before, we have been interested in the KPN model of computation for data flow driven tasks. In the Kahn paradigm, tasks communicate with each other via unbounded FIFO channels. Reading from channels is done in a blocking manner, while writing is non-blocking. To model channels, we use SysML [2] flows stereotyped by *"Dchannel"* for data channels or *"Cchannel"* for control channels. Each data channel has one tagged value that is the number of available tokens on the channel. Control channel has one tagged value that is the number of available events. In order to model top level module (i.e. application), we introduce a new stereotype called *Top*. We note that when a child port is the same as its parent port, we do not model the channel (see figure 3).

To model internal behaviours of data flow driven tasks, we use UML2.0 activity diagrams with CGAs. Each CGA belongs to one of the three generic types: Computation Actions (CAs), Read Actions (RAs), or Write Actions (WAs). For this purpose, we define a new stereotype called "*Compute*". We use this stereotype to model CGAs computations. It contains two tagged value that specify the number of elementary instructions inside a computation, and the type of the elementary operation (i.e. integer or float). Each RA or WA has two arguments: the name of the port and the size of the transmitted data expressed in term of token numbers. Write action is modeled via UML2.0 Send action. Control driven tasks are modelled as finite state machine (FSM) using statecharts.. Transitions and states take zero time.

In order to estimate application performance, hardware architecture should also been abstracted. So components of the platform are modeled as an abstraction of their fine grain models and they are generic enough so that the whole architecture could potentially be used for modeling all types of hardware platforms. Here, we identify four generic components that are CPU, bus, bridge, and RAM models. The CPU model is characterized by the Task switching time: the time to go to the "idle" state, the number of cycles for an elementary operation, the average amount of power consumed per cycle in the running mode, the average amount of power consumed per cycle in the idle mode, and the scheduling algorithm. Using UML, we define a stereotype called "CPU" with the above mentioned parameters as tagged values.

A bus access is needed only for performing a write operation. Read operation does not require bus access because a CPU reads data from the FIFO in its local interface. However, if the communication between tasks is done via RAM, read operation requires bus access. The Bus model is characterized by the bus type which can be shared or dedicated, the transfer rate expressed in term of tokens number per cycle, the amount of power consumed per cycle in the running mode, and the arbitrage policy in the case of a shared bus. We define a stereotype called "BUS" with the above mentioned parameters as tagged values.

The RAM is characterized by its reading rate, writing rate, the amount of power consumed per cycle in the running mode. We define a stereotype RAM with the above mentioned parameters as tagged values.

We use the bridge to connect multiple buses. The bridge is characterized by its transfer rate, the amount of power consumed per cycle in the running mode. We define a stereotype BRIDGE with the above mentioned parameters as tagged values.

Mapping consists in allocation and scheduling of application components to architecture components, so tasks are mapped to CPUs, and abstract channels are mapped to either buses or RAMs. We define a new stereotype called *"AllocatedTo"*. This stereotype is applied on the UML constraint and it has two stereotypes. The first one specifies the name of the hardware component to which logical component should be allocated. The second one designates a number that determines the execution order of the task (the transfer). We note, that if there is only one path between two hardware components, then we do not have to specify the transfer mapping and the path (i.e. bus) is determined from the architecture. By default, all transfers are mapped to buses unless, the designer introduces the transfer mapping via RAMs or dedicated buses explicitly.

IV. MAPPING OF UML MODELS TO MAUDE

Using Maude, each data flow driven task is specified as follows: *< A: Dtask | hwname: cpu, state: sta, action: act >* . where *A* is an instance of the *Dtask* class; *hwname* is the name of the CPU to which the task *A* is mapped; *State* is the current state of the task (ready, run, wait, and idle); *Action* designates the CGA performed by the task. Here, we have three cases according to the type of *Action*: If *Action* is a compute CGA, then we add the attributes *Number*, and *Type. Number* designates the number of elementary operations involved in a compute action. *Type* is used to specify the type of elementary operations (i.e. Integer or float). If *Action* is an input/output CGA, then we add the attribute *Token.* The latter determines the number of transmitted data tokens (read/write) over the channel. Finally, *Action* can be a waiting action (i.e. waiting for an event occurrence). Control driven task is declared as: *< A: Ctask | hwname: cpu, state: sta, FSMS: fsms >* where *FSMS* can designate the name of the FSM current state or a waiting state (i.e. waiting for an event occurrence). Similarly, each abstract channel is declared either as a class named *Dchannel* or *Cchannel*: *< ch : Dchannel | hwname : hw, source : A, target : B, available : x >* or *< ch : Cchannel | hwname : hw, source : A, target : B, size : x >* where *ch* is the name of the abstract channel; *Hwname* designates the name of the bus or the RAM to which the channel is mapped.; *source* and *target* designate the source and the target tasks linked by *ch*; *available* designates the current data tokens available on the data channel FIFO or the RAM, and *size* designates the number of available events on the events FIFO attached to control channel.

Using Maude, we declare each CPU as a class with a set of attributes: *< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >* where *cpu* is the name of the processor; *LinkTo* designates the name of the bus to which *cpu* is connected. In the case where *cpu* is connected to more than one bus, we have to add many *LinkTo* attributes. *ContextSwitch* designates the overhead due to tasks context switching; *GoIdle* designates the cycles number required for transition to the idle state; *Iop*, and *Fop* designate the cycles number required for the execution of an integer and float elementary operation respectively; *Power*, and *PowIdle* designate the amount of power consumed per cycle in a run state and an idle state respectively, and finally *TPower* is the total power consumed by the CPU.

TABLE I. CORRESPONDENCE BETWEEN UML AND MAUDE

UML	Maude
Class	Class
SysML Flow	Class
Tagged value	Attribute
CGA	Attribute
FSM state	Attribute
Activity diagram transition	Rewriting rule
FSM transition	Rewriting rule

We declare the bus as a class named BUS *<bus : BUS | Speed : sp, Power : pb, TPower : tpb, free : true >* where *bus* is the name of the bus instance; *Speed* designates the transfer rate of the bus, and *free* is a Boolean variable. When it is true, *free* indicates that the bus access is granted by a task, otherwise it is false. The attribute *free* is used only with shared bus. When more than one a task requests a bus access at the same time, a scheduling policy must be defined. In our case we adopt a FIFO policy.

We declare RAM as a class named RAM *<ram : RAM | LinkTo : bus, Rrate : rr, Wrate: wr, Power : pb, TPower : tpr >* where *ram* is the name of the RAM instance; *Rrate* and *Wrate* designate the reading and writing rates respectively.

Bridge is declared as a class named BRIDGE *<bridge : BRIDGE | LinkTo :bus1, LinkTo : bus2, Speed, Power : pb, TPower : tpb >* where *bus1* and *bus2* designate the two buses to which *bridge* is connected.

Using Maude, the data flow driven task behavior is specified as a sequence of rewriting rules. Each rewriting rule corresponds to a state transition that triggers the execution of a CGA. Rewriting rules can be conditioned by the occurrence of some events and/or conditions eventually. For the sake of the space, we give some rewriting rules. Assume that task *A* is mapped to *cpu.*

```
rl [start] : ***1
start(A)
< A : Dtask | hwname : cpu, state : ready, > =>  < A : Dtask |
hwname : cpu1, state : run, action : readC, token : 5 > .

crl [readCwait] : ***2
< A : Dtask | hwname : cpu, state : run, action : readC, token : n >
< cpu : CPU | LinkTo : bus, ContextSwitch: cont, GoIdle : idl, Iop
: iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp > < ch1 :
Dchannel | hwname : bus1, source : A, target : B, available : x >
< C : Dtask | hwname : cpu, state : s >
=>  < A : task | hwname : cpu, state : wait, action : readC, token :
n – x > < cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle :
idl,Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp +
(float(cont) * pw) > < ch1 : Dchannel | hwname : bus1, source :
A, target : B, available : 0 > < C : Dtask | hwname : cpu, state : s
> wakeup(C) if (x < n) and (s == ready) .
```

```
crl [EV5occurence] : ***3
< A : Dtask | hwname : cpu, state : run, action : waitEV5 >
< ch : Cchannel | hwname : bus1, source : A, target : B,  size : sz
> => < A : Dtask | hwname : cpu, state : run, action :
computeDCT, Nombre : 1000, Type : integer > < ch : Cchannel |
hwname : bus1, source : A, target : B,  size : sz – 1 > if sz > 0 .

rl [computeDCT] : ***4
< A : Dtask | hwname : cpu, state : run, action : computeDCT,
Nombre : 1000, Type : integer >
< cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop
: iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
=> < A : Dtask | hwname : cpu, state : run, action : writeA, token
: 5 > < cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle :
idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp +
(float(1000 * iop)* pw) > .

rl [AccessBus] :***5
< A: Dtask | hwname : cpu, state : run, action : writeA , token : n
> < bus: BUS | Speed : sp, Power : pb, TPower: tpb, free : f >
=> if f == true then < A : Dtask | hwname : cpu, state : run,
action : writeA , token : n> <bus : BUS | Speed : sp, Power : pb,
TPower :tpb, free : false > AccessBus(bus) else < A : Dtask |
hwname : cpu, state : wait, action : writeA , token : n > < bus :
BUS | Speed : sp, Power : pb, TPower : tpb, free : f >  fi .

rl [writeA] : ***6
AccessBus(bus)
< A : Dtask | hwname : cpu, state : run, action : writeA , token : n
> < ch2 : Dchannel | hwname : bus, source : A, target : D,
available : x > < bus : BUS | Speed : sp, Power : pb, TPower :
tpb, free : f >
=> < A : Dtask | hwname : cpu, state : run, action : computeINV,
Nombre : 200, Type : float >  < ch2 : Dchannel | hwname : bus,
source : A, target : D, available : x + n > < bus : BUS | Speed :
sp, Power : pb, TPower : tpb + (pb * (float(n) / sp)) , free : true > .
```

Rule 1 forces the task A to change its state from *ready* to run the first CGA namely *readC* (read five tokens). We use the *start* method to specify the first task to be executed by *cpu*. Rule 2 forces the task *A* to wait on the read operation if the number of available tokens on the input channel is less than read tokens. In this case, *cpu* stops *A* and wakes up the next task named *C* which is ready using the *wakeup* method. During this context switching, we update *TPower*. In general when a task gets blocked (i.e. read operation or waiting for an event), *cpu* wakes up the next ready or blocked task (the order of execution is extracted from UML mapping).

Rule 3 specifies the occurrence of event *ev5* that permits to task *A* to perform *computeDCT* including 1000 integer operations.

Rule 4 specifies the execution of *computeDCT* and the updating of *TPower*.

Rule 5 permits the task *A* to grant the shared bus access for data writing: if the bus is free (free == true), then the access is granted, otherwise, task *A* will be blocked.

Rule 6 specifies the writing operation and corresponding power update. Since writings are not blocking, we have no condition on the maximum size of data FIFOs. The task releases the bus when it terminates writing.

To specify control driven tasks behaviours, we apply the same principles as for data flow driven tasks. However, instead of dealing with computation actions, and data inputs/outputs, we are focused on task states and associated input/output events.

V. PROPERTIES SPECIFICATION AND VERIFICATION

At this level of abstraction, we can verify some undesirable or/and desirable properties as deadlock. The latter can occur due to an improper scheduling. Using the Maude command *"search in application: initial =>! X:conf such that TaskEnd(X:conf) == true."*, we can easily verify whether all tasks reach the idle state (that means there is no deadlock). Note that the command *"search S =>! S', where =>!"* means that we are looking for terminal states, that is, states from which no further rewritings can take place. *TaskEnd* is a function defined as:

```
sort conf .
subsort conf < Configuration .
op sta : conf -> states .
op TaskEnd : conf -> Bool .
eq sta (< T1 : Dtask | hwname : cpu, state : st >) = st .
eq sta (< T2 : Ctask | hwname : cpu, state : st >) = st .
eq TaskEnd (T) = if sta(T) == idle then true else false fi
```

Similarly, we can verify whether the bus is always busy which is a non-desirable property. For this reason we use the command *"search in application : initial =>! X:conf such that BusBusy(X:conf) == true ."* where *BusBusy* is a function defined as:

```
op BusBusy : conf -> Bool .
eq BusBusy (< bus : BUS | Speed : sp, Power : pb, TPower : tpb,
free : bool >) = if bool == false then true else false fi
```

Another important property we can verify is the fact that the amount of data tokens buffered in data channels FIFO does not exceed a certain threshold in every state of the system. For this reason, we define a property named *FIFOsize* defined as:

```
var cf : configuration .
op FIFOsize : Configuration -> Prop .
ops ch A B bus : -> Oid .
vars x TS : Nat .
ceq < ch : Dchannel | hwname : bus, source : A, target : B,
available : x, threshold : TS > cf |= FIFOsize (< ch : Dchannel |
hwname : bus, source : A, target : B, available : x, threshold : TS >
cf) = true if x < TS .
```

Using the command: *"red modelCheck(initial, []FIFOsize(initial)) ."*, we can easily verify the FIFOsize property. *initial* is the name of the initial configuration. The symbol [] means globally: The property must hold for every state of every computational path. If the property is not true, a counterexample is given.

VI. EXAMPLE

An example is presented in this section with the objective to demonstrate the abstraction level of application tasks, and the formal verification. Task *A* performs a CGA *computeA1* including 300 integer operations, and writes 10 tokens of data over *ch1*. Then, it waits for *request1* event from *Controller1* (via *ch3*) to perform a second CGA *computeA2* including 120 float operations before it terminates. Task *B* performs a CGA *computeB1* including 100 integer operations and attempts to read 5 data tokens from *ch1*, then it performs *computeB2* including 50 float operations, and writes 8 data tokens over *ch2* before its termination. Task *C* attempts to read 5 data tokens from *ch2*, performs a CGA *computeC* including 1000 integer operations, then it sends an event *ok1* to *Controller2* over *ch5*, and terminates (see figure 4). Figure 5 shows UML statecharts of *Controller1* and *Controller2*. Before execution starts, we assume that the number of available tokens for *ch1* is equal to 0, and 7 for *ch2*.

To accomplish our objective, we have used Rhapsody 7.2. [6]. Maude code is generated as text a file from Rhapsody automatically due to its VB (Visual Basic) interpreter. The result of the rewriting for our example is shown in figure 6.

VII. RELATED WORKS

With regard to the application of UML to the ESs field, the literature is very rich. However we can mention some pertinent works [2]. MARTE is an UML2.0 profile that targets real time embedded systems. It offers a facility for modeling, and analyzing real time applications. TUT profile provides an automated path from UML design entry to FPGA prototyping including the functional verification and the automated design exploration focusing on automatic profiling and performance values back annotation. Gaspard2 is an UML2.0 profile, targeting intensive signal processing (ISP) domain. It is based on the ISP profile. The latter allows the expression of task and data parallelisms using Array-OL language UML-PLATFORM is an UML profile for wireless communication protocols. It is based on the Metropolis meta-model. DIPLODOCUS is an UML profile for SOC LOTOS-based formal verification and quick SystemC simulation. Power consumption estimation issue is not addressed. The main problems with most of these profiles are the lack of formal support for early analysis and validation and high level power consumption estimation.

VIII. CONCLUSION

In this paper, we present our approach for ESs high level modeling using UML. Formal verification of some properties and power consumption estimation are accomplished by mapping UML models to rewriting-based Maude language. We have used the Rhapsody environment for both modeling and code generation. As a perspective, we plan to enrich our proposed model for control and data driven tasks by adding some realistic information concerning time and power consumption.

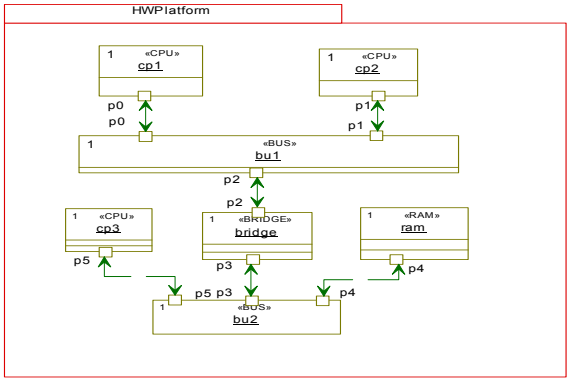

Figure 2. UML architecture modeling

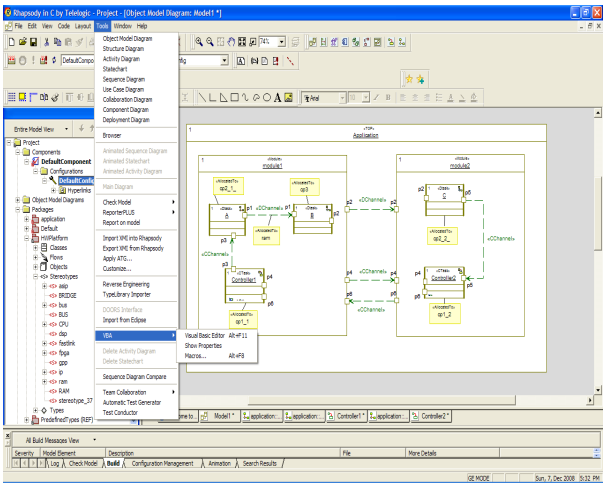

Figure 3. UML application and mapping modeling

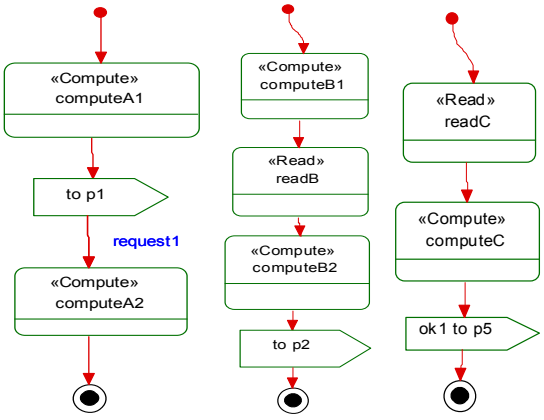

Figure 4. UML data flow driven tasks internal behaviors modeling

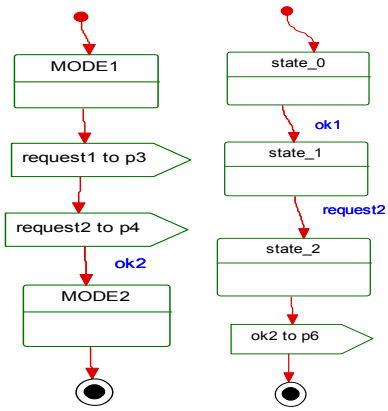

Figure 5. UML control driven tasks internal behaviors modeling

Figure 6. Rewriting results

REFERENCES

[1] B. Kienhuis, F. Deprettere, P.V. Wolf, and K.Vissers, *A Methodology to design Programmable Embedded Systems*: *The Y-chart approach*. In LNCS series vol. 2268, page 18-37 by Springer Verlag © 2001.

[2] F. Boutekkouk, M. Benmohammed, S. Bilavarn, M. Auguin, UML2.0 profiles for Embedded Systems and Systems On a Chip (SOCs). In *JOT (Journal of Object Technology)*, January 2009.

[3] G. Booch, J. Rumbaugh, I. Jacobson, *Unified Modeling Language User Guide* (Addison-Wesley, 1999).

[4] J. Meseguer, *Rewriting as a unified model of concurrency*. In proceedings of the Concurr'90 Conference, Amsterdam, pp. 384-400, Springer LNCS Vol. 458, 1990.

[5] T. McCombs, *Maude 2.0 Primer, Version 1.0. International report*. SRI. International 2003.

[6] www.ilogix.com