COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems

Xu Ke, Krzysztof Sierszecki, Christo Angelov
Mads Clausen Institute for Product Innovation, University of Southern Denmark
Grundtvigs Alle 150, 6400 Soenderborg, Denmark
{xuke, ksi, angelov}@mci.sdu.dk

Abstract

The paper presents a generative development methodology and component models of COMDES-II, a component-based software framework for distributed embedded control systems with real-time constraints. The adopted methodology allows for rapid modeling and validation of control software at a higher level of abstraction, from which a system implementation in C can be automatically synthesized. To achieve this objective, COMDES-II defines formally various kinds of components to address the critical requirements of the targeted domain, taking into consideration both the architectural and behavioral aspects of the system. Accordingly, a system can be hierarchically composed from reusable components with heterogeneous models of computation, whereas behavioral aspects of interest are specified independently, following the principle of separation-of-concerns. The paper introduces the established generative methodology for COMDES-II from a general perspective, describes the component models in details and demonstrates their application through a DC-Motor control system case study.

1. Introduction

Recently emerging concepts and techniques, such as *domain-specific modeling* (DSM) and *component-based development* (CBD) are considered particularly appropriate for the efficient development of reliable embedded software systems [1, 2]. DSM provides intuitive modeling concepts which are familiar to the domain experts, and accommodates important domain-specific characteristics of embedded systems, such as system concurrency, environmental physicality, real-time operation, etc. CBD can be regarded as one of the most suitable design paradigms for domain-specific modeling. Due to the inherent benefits brought by reusability of components, and higher-level of abstraction (modeling systems rather than programming systems), an embedded software system can be efficiently constructed from prefabricated and reusable components. Moreover, from a software engineering point of view, CBD is also an effective way to bridge the gap between the conceptual system design models and the concrete system implementation [3], provided that an automatic code generation technique is developed.

The above arguments have been carefully taken into consideration during the development of COMDES-II (COMponent-based design of software for Distributed Embedded Systems - version II) [4], a component-based software framework intended for efficient development of distributed embedded control systems with *hard real-time* requirements. The framework attempts to establish an engineering methodology encompassing high-level modeling and analysis issues as well as low-level implementation and deployment techniques. As a result, it emphasizes the following development issues:

- Component models of computation (MoC) and the associated modeling techniques that can be used to specify significant characteristics of real-time control systems, following the principle of *separation-of-concerns*.
- Practically applicable analysis techniques aiming at verifying system behavior at a higher abstraction level, e.g. model-checking.
- Advanced algorithms and data structures enabling reusability and re-configurability of components.
- Automatic code generation techniques to maximally reduce the manual coding effort, hence minimizing the errors introduced by manual coding.
- Proper compilation and configuration techniques, which can be used to automatically synthesize the deployable systems from prefabricated component executables.

This paper presents the COMDES-II design methodology, followed by a detailed description of the framework component models and the associated modeling techniques. The rest of paper is organized as follows: Section 2 outlines the COMDES-II methodology from a general perspective in order to provide a panorama of the framework features. Sections 3 and 4 present the framework component models in details, highlighting the architectural and behavioral aspects, respectively. Section 5 presents a DC-Motor control system case study developed using COMDES-II to demonstrate the application of the framework. Section 6 presents related research. Finally, the concluding section summarizes the features of the framework and their implications.

2. COMDES-II methodology

In order to realize efficient and unambiguous development of reliable control software systems, COMDES-II employs a *generative programming* software engineering methodology, which can be used to automate the generation of system implementations from higher-level abstractions represented as textual or graphical models [5]. This approach is characterized by a domain-specific, model-driven architecture in which each abstraction layer is represented as a set of models with distinct abstraction levels, as shown in Figure 1.

The topmost abstraction layer, which formalizes the targeted domain characteristics, and the meta-modeling layer defining the framework modeling language are located in the problem space of this methodology, in the sense that the concrete models and techniques associated with these layers, e.g. mathematical formalisms and corresponding meta-models, should be developed by the framework developers. The other three layers: domain-specific modeling, implementation and deployment layers are in the solution space because the corresponding modeling and development automation techniques, belonging to these layers, are solutions provided by the framework developers to the framework users to solve their application-specific problems in a top-down abstraction refinement process. In this context, the development of a system starts from the specification and verification of system models, through code generation and validation, down to the configuration of system deployable code (executable systems at low abstraction level).

The entire stepwise refinement process of system abstraction levels is fully assisted by the formally defined models and the corresponding model-based automation techniques, resulting in unambiguous and rapid development of reliable software systems [6].

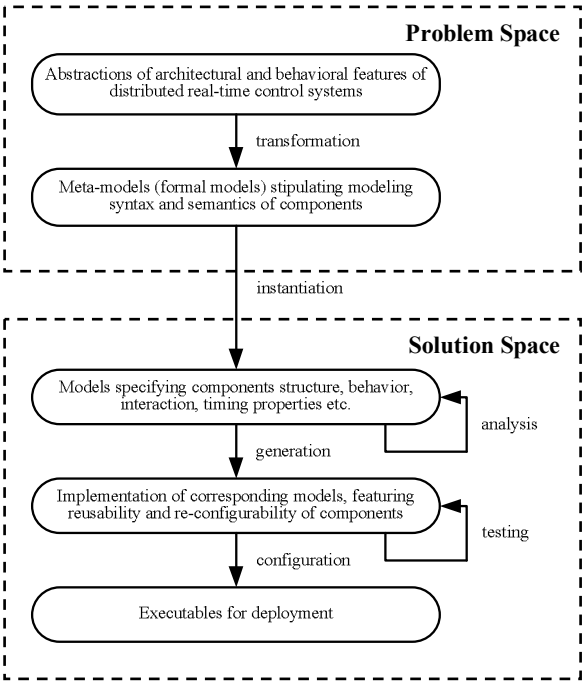

Figure 1. Generative programming methodology for COMDES-II

3. Architectural modeling of systems

COMDES-II employs a hierarchical model to specify system architecture: at the system level a distributed control application is conceived as a network of communicating *actors* (active components). Distributed actors interact transparently with each other by exchanging labeled messages (signals), following an asynchronous producer-consumer protocol known as content-oriented message addressing.

At the actor level, an actor is specified as a software artifact containing multiple *I/O drivers* and a single *actor task* (execution thread). I/O drivers are classified as *communication drivers* and *physical drivers*, which are associated with the actor task through a dataflow relationship denoting the exchange of local signals.

The I/O drivers are responsible for sensing or actuating signals from/to network or physical units, while the actor task processes the acquired signals to fulfill the required functionality which is specified by a composition of different *function block instances.* Function block instances are instantiations of reusable and reconfigurable function block *types*, which can be categorized into four function block *kinds* (meta-types): *basic*, *composite*, *modal* as well as *state*

machine function blocks. A detailed description of each kind of function blocks will be given in Section 4.

The example in Figure 2 shows the presented hierarchy of components used in modeling CODMES-II systems. This architectural model features openness and information hiding at component level, which brings typical benefits, such as succinctness and readability, to the COMDES-II systems architecture.

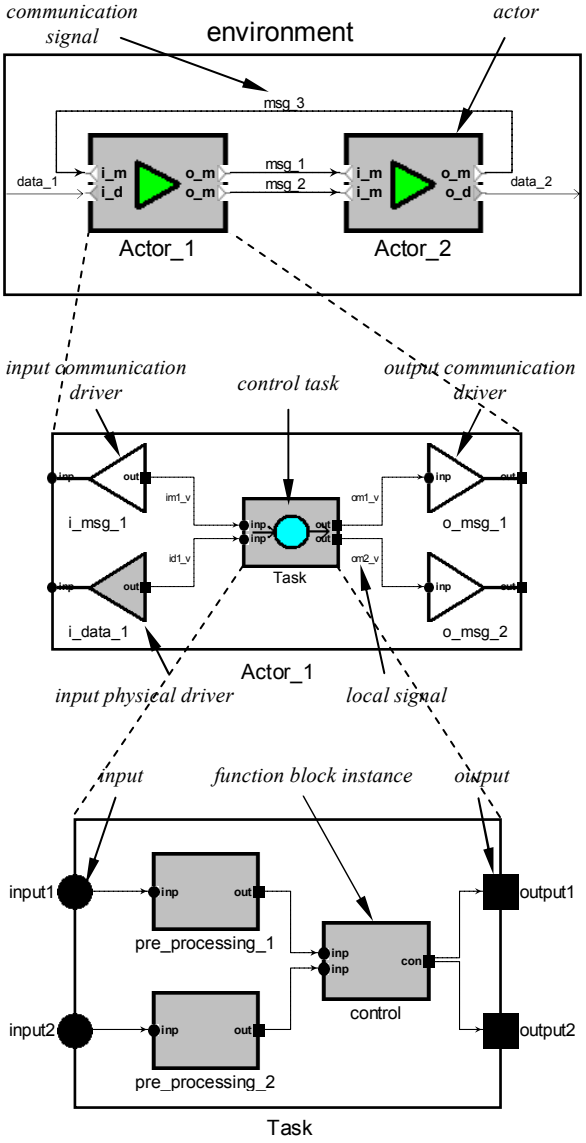

Figure 2. Hierarchical architecture model of a COMDES-II system

4. Behavioral modeling of systems

In order to clearly model various behavioral properties, COMDES-II adopts a *separation-of-concerns* approach to systematically decompose different behavioral concerns into separate modeling units.

4.1. Actor model

In COMDES-II, all non-functional information is specified with respect to actors, including *physicality*, *real-time reactivity* and *concurrency*. The operational semantics concerning these non-functional issues is completely regulated by the underlying real-time kernel $HARTEX_{TM}$ [1] which employs a fixed-priority timed multitasking scheduling policy [7, 8].

An actor is activated by an *execution trigger*, which may be a periodic or sporadic event. The occurrence of a triggering event is handled by the kernel to notify the corresponding actors for activation. Upon activation, the input drivers of the actor are executed atomically to acquire all input signals, which will be latched throughout the whole actor execution. The execution trigger releases the internal actor task, which will process the obtained input data once it becomes the highest priority task among all released/preempted tasks in the processor. The processed data will be buffered into output drivers that can be atomically executed to generate output signals when the corresponding actor *deadline* expires. In this computation model, preemption of actor task operation does not result in actor output jitter, as long as the task can finish its operation before the deadline. The *split-phase* execution pattern of COMDES-II actors is illustrated as in Figure 3.

Obviously, COMDES-II actors have a typical *read-do-write* operational semantics, in which the *read* action only takes place at the activation instant and no new input signals will be read during actor task operation (*do* action). As a result, the *do* action of an actor is synchronous, i.e. the task operation takes zero *logical execution time* (LET), because the computation result generated by an actor task principally reflects the actor status at the activation instant. These properties are vital for establishing the correct analysis models for COMDES-II systems.

The timed multitasking model of computation separates the timing behavior of an actor from its functional aspect, which may facilitate timing analysis, since real-time properties can be checked using schedulability analysis, rather than being checked together with the functional behavior (e.g. using timed automata models).

[1] $HARTEX_{TM}$ is a hard real-time kernel for embedded systems developed by the Software Engineering Group, Mads Clausen Institute for Product Innovation, University of Southern Denmark.

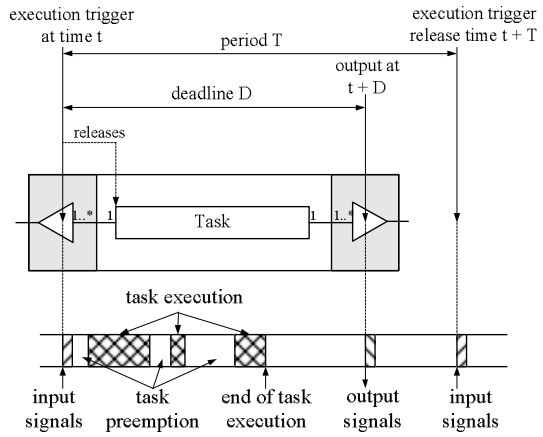

Figure 3. Split-phase execution of actors under timed multitasking

The combination of timed multitasking with transparent signal-based communication has resulted in a novel operation model - *Distributed Timed Multitasking* (DTM) [7], which is one of the distinctive features of COMDES-II. It provides for jitter-free execution of periodic distributed actor transactions, whereby transaction input and output actions are executed at precisely specified time instants on the time axis. A detailed description is referred to [4] and [7].

4.2 Function block models

Function blocks (FBs) are pure functional components implementing concrete computation or control algorithms, which can be used to specify the system functional behavior. COMDES-II defines four *kinds* of FBs: *basic*, *composite*, *modal* as well as *state machine* FBs to help specify various kinds of system functionality, in which basic and composite FBs can be used to model continuous behavior (data flow), while state machine and modal FBs describe the sequential system behavior (control flow). Moreover, COMDES-II provides adequate modeling and implementation techniques integrating heterogeneous data flow and control flow models to realize hybrid (modal continuous) system operation [4]. Each *kind* of FB may have a number of FB *types*, which can be instantiated into concrete FB *instances* with different settings. FB instances are the actual building blocks incorporated into the specific host actors to realize the required system functionality.

4.2.1 Basic FB models. Basic FBs are the elementary function blocks in COMDES-II, from which more sophisticated kinds of FBs can be constructed, such as composite FBs and modal FBs. The design model of a basic FB is shown as in Figure 4.

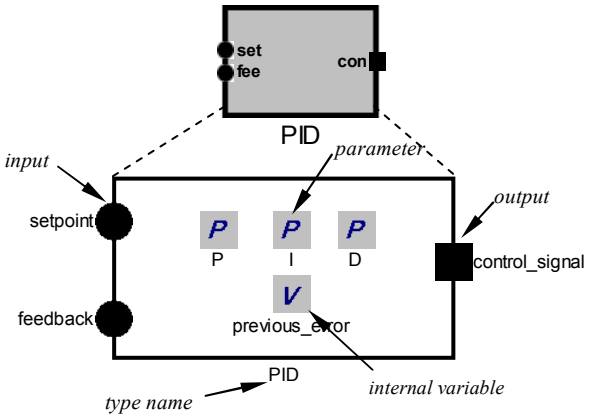

Figure 4. An example of basic FB

The presented basic FB model highlights the specification of FB attributes, in which finite sets of inputs, outputs, parameters and internal variables are declared, whereas the functionality is uniquely determined by the corresponding signal transformation or control functions (e.g. PID). The attribute aspect of all kinds of FB models is defined in a similar manner; therefore the following presentation of FB models will only focus on the specific functionality aspects of complex FB kinds.

4.2.2 Composite FB models. A *composite* FB is a composition of basic and/or other composite FB instances used to accomplish complex computational behavior. The *functionality aspect* of a composite FB is represented as a function block diagram consisting of interconnected FB instances (see Figure 5), which essentially employs a *synchronous data flow* (SDF) model of computation.

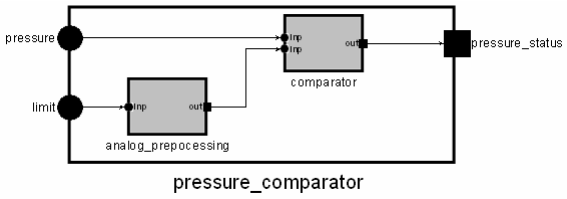

Figure 5. An example of composite FB

The execution sequence of internal FB instances in a composite FB is controlled by a static execution schedule, which is a data structure automatically

synthesized from the corresponding FB diagram, according to the signal flow from inputs to outputs. An assumption for deriving such an execution schedule is that the contained FB diagram is acyclic, because the feedback connections in a synchronous function block diagram may cause a causality problem.

4.2.3 State machine and modal FB models. In COMDES-II, state machine FBs and modal FBs are jointly used to specify the system sequential behavior, as illustrated in Figure 6.

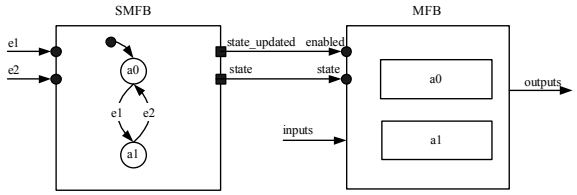

Figure 6. Interaction between a state machine FB and a modal FB

In this context, a state machine FB consists of a number of *binary event/guard* inputs, an *event-driven* state machine model, and exactly two outputs: *state* and *state_updated* (see e.g. Figure 7). The internal state machine model contains a dummy initial state pointing to the actual initial state of the machine, a graphical label with the name history meaning that the state machine is historic, a number of states and state transitions that are labeled by events, guards and transition orders. Transition events/guards are manipulated as state machine FB input signals acquired from input drivers or preprocessing FBs, and transition order is a number indicating the importance of the transition, i.e. which transition should be fired when multiple transition triggers associated with the current state are evaluated as true (transitions are evaluated starting from 1). This technique avoids the undesirable non-determinism of state transitions, as required in safety-critical control systems. When a state machine FB is executed, the internal state machine model parses the binary event/guard input signals, determines the current state and updates two outputs: *state* and *state_updated*, where *state* represents the currently active state, and *state_updated* is set *true* if a state transition has happened, otherwise it is *false*.

The two output signals from a state machine FB are used by the corresponding modal FBs to execute the control actions associated with the specific state (see Figure 6), and one state machine FB (master) may control multiple modal FBs (slaves) if these modal FBs share an identical state machine structure. As an example, Figure 8 illustrates the internal structure of a modal FB. A modal FB may contain multiple inputs and outputs, which can be connected with the I/O interfaces of the internal operation modes (states) via dataflow connections. Operation modes are software artifacts, within which the concrete control actions are specified by means of the corresponding function block diagrams. The selection of an executing operation mode is decided by the currently active *state* information provided by the supervisory state machine FB, whereas the execution of an operation mode is enabled by the *state_updated* value, i.e. the control action should be performed only when a state transition occurs, since the state machine model is event-driven.

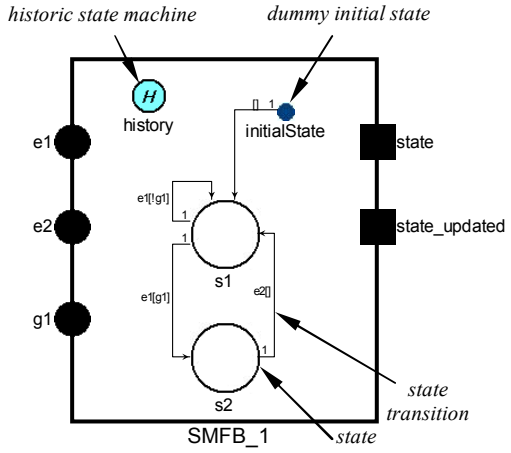

Figure 7. An example of state machine FB

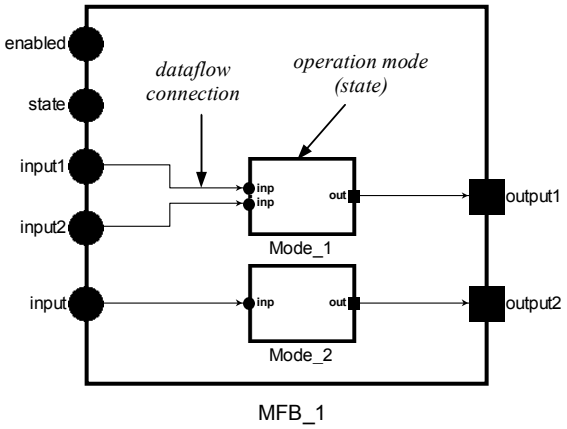

Figure 8. An example of modal FB

Modal FBs support hierarchical composition, i.e. an operation mode of a modal FB can be refined by another state machine FB instance and the related slave MFB instances. This feature enables a hierarchical

state machine structure and a compositional semantics for modal FBs. Heterogeneous composition is also possible by allowing the integration of function block diagrams implementing continuous control behavior into the corresponding sequential modes of operation. Additionally, heterogeneity also means that a state machine FB instance and its slave modal FB instances can be connected with other basic and/or composite FB instances, resulting in a hybrid control system.

5. DC-Motor control system case study

The presented framework has been experimentally validated through a number of case studies, including the Production Cell Case Study, the Steam Boiler Control Specification Problem and the DC-Motor Control System Case Study. A tutorial version of the latter is presented in this section in order to illustrate the application of the framework.

The system is a distributed hybrid control application. It controls a DC-Motor, through a real-time network (CAN). The control commands are released from an operator station that interacts with the user through a keypad and an LCD display. The three nodes: Sensor, Controller, and Actuator implement the control loop, i.e. control the speed and direction of the motor.

At the system level the distributed control system is modeled as an actor diagram consisting of three actors: Sensor, Controller, and Actuator, which interact with the DC-Motor and operator station (not shown here) as depicted in Figure 9.

DC-Motor and operator station

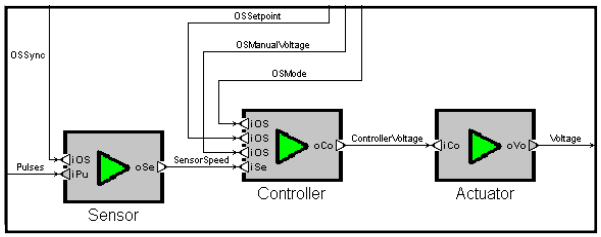

Figure 9. DC-Motor control system

The Sensor actor accepts Pulses from an optical encoder indicating the current speed of the motor, and sends the information via the SensorSpeed message to the Controller actor. Depending on the data from operator station (messages: OSSetpoint, OSManualVoltage, OSMode), the Controller computes control signal (ControllerVoltage) and sends it to the Actuator, which applies Voltage to the motor. Control system actors are executed in a phase-aligned transaction triggered by an OSSync message issued periodically by the operator station (Figure 9).

The rest of the discussion focuses on the Controller actor, whose structure is presented as in Figure 10.

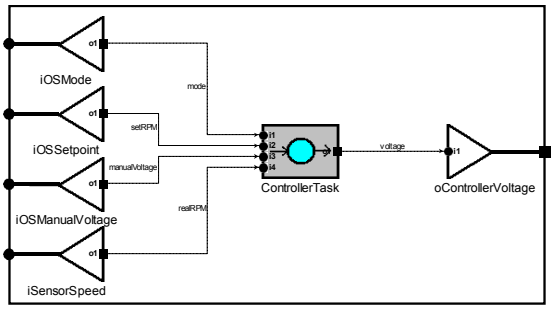

Figure 10. Controller actor

The internal structure of the ControllerTask is modeled as a composition of function blocks as shown in Figure 11, in which the CStateMachine and CModal FBs are responsible for switching to appropriate mode of operation, i.e. manual or automatic control, and executing the corresponding control algorithms.

The CStateMachine FB in the ControllerTask has three states: *init*, *manual* and *automatic*, representing the possible status that the Controller actor could be in. The required state transition signals (manual, automatic) used by the CStateMachine FB are computed beforehand by two comparator FB instances: compareModeManual, compareModeAutomatic. Outputs generated in different modes (*init*, *manual*, *automatic*) of the modal FB CModal are multiplexed by the multiplexVoltage multiplexer FB, which will write correct output signals to the corresponding output buffer according to the *state* information provided by the CStateMachine FB.

The control action performed in the automatic mode of the CModal FB is presented in Figure 12. This operation mode implements a modal PID control action, which exhibits the hybrid (modal continuous) behavior of the system. In this operation mode, the determination of direction status is managed by the AutStateMachine FB containing *stop*, *leftControl* and *rightControl* states. The corresponding modal FB AutModal performs the continuous PID control actions associated with the *leftControl* and *rightControl* states, respectively. Once again, outputs generated in different modes (*stop*, *leftControl* and *rightControl*) of the modal FB AutModal are multiplexed by the multiplexVoltage multiplexer FB, which filters out the correct output according to the active *state* information provided by the AutStateMachine FB.

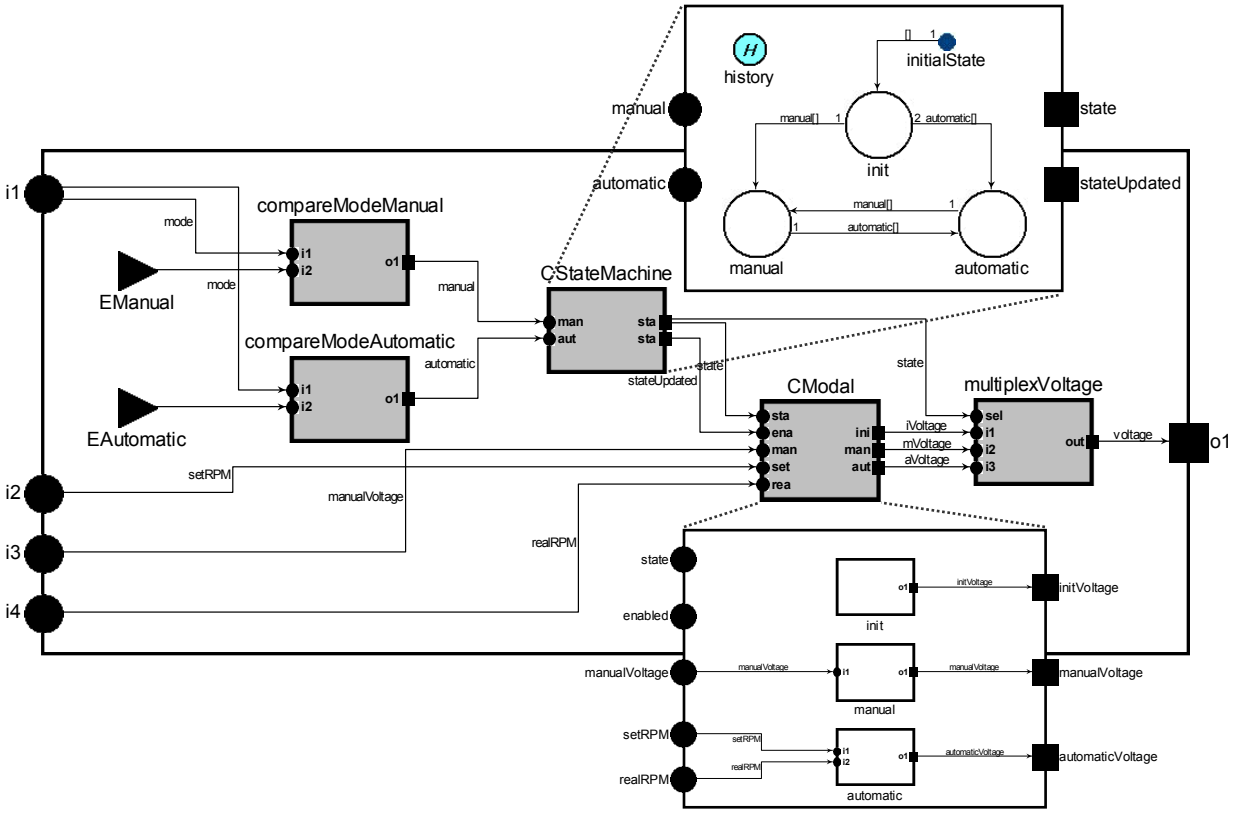

Figure 11**. Internals of the ControllerTask**

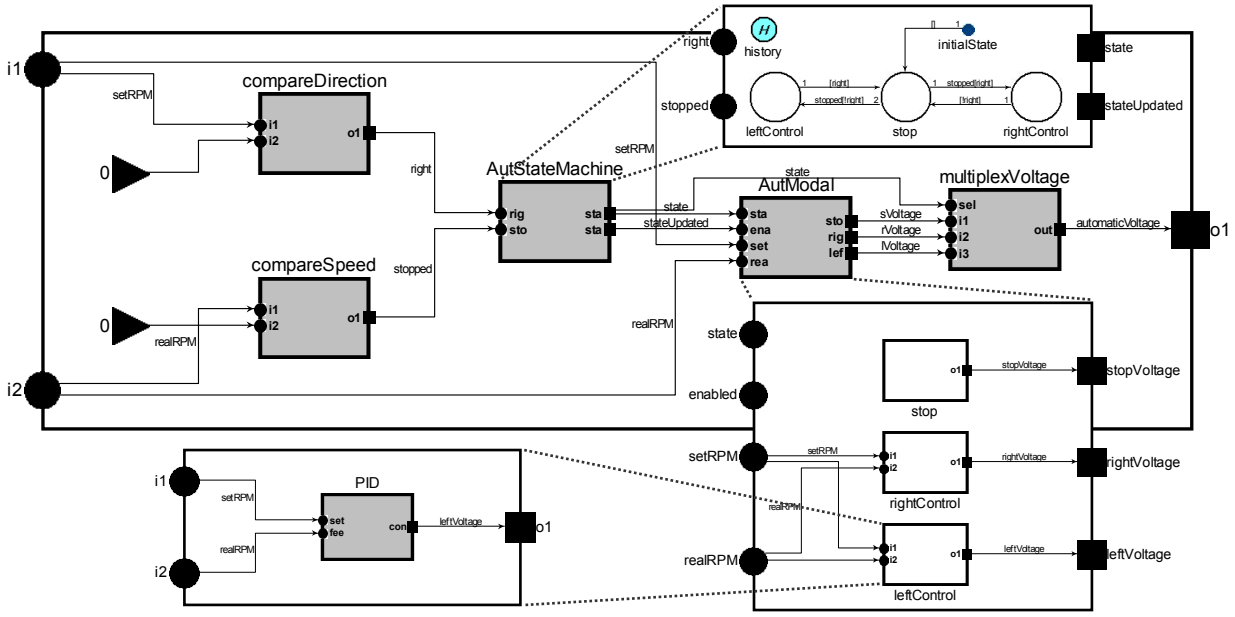

Figure 12**. Internals of the automatic mode in CModal FB**

6. Related research

Actor-based models are highly popular in the Software Engineering domain. A great number of frameworks use port-based objects that are mapped onto real-time tasks, e.g. *Timed Multitasking (TM)* [8] and *xGiotto* [9]. Unfortunately, it is difficult to identify reusable components with those models, since reusability is generally not well manifested at the task level. It is usually identified at a lower level exemplified by finer-grain components, such as function blocks and their equivalent - passive port-based objects. However, their use may result in complex (nested) models featuring too many ports and port connections.

Industrial software standards like IEC1131-3 and IEC 61449 reduce model complexity by adopting domain-specific modeling and interaction techniques, i.e. function block diagrams and signal-based communication. However, these frameworks do not provide modeling techniques and component definitions at task and subsystem levels, and component hierarchy as well as heterogeneity are not well specified either.

These limitations are overcome in *COMDES-II* by introducing units of concurrency, i.e. actors, in addition to function blocks - an application is composed of actors and actors are configured from function blocks, whereby signal-based communication is used at all levels of specification. Moreover, transparent signal-based communication is combined with the concept of timed multitasking, resulting in an extended model denoted as *Distributed Timed Multitasking*. The latter has been inspired by the original *TM* model [8], and is similar to the computational model of the *xGiotto* framework [9]. However, these frameworks use port-based tasks and the reported implementations seem to be limited to single-computer systems, whereas *COMDES-II* accommodates distributed actors that interact via signal-based communication. On the other hand, *TM* and *xGiotto* employ only task-level actor models and do not define reusable components, such as function blocks. Conversely, *COMDES-II* specifies a number of generic function block kinds that provide support for a broad of sequential, continuous as well as heterogeneous (hybrid) systems.

7. Conclusion

The paper has presented a *generative* methodology and the associated systems design philosophy of COMDES-II, a component-based software framework for distributed, real-time embedded control systems.

The adopted methodology is anticipated to automate the implementation of system and component codes from the corresponding high-level design models. This generative approach may remarkably leverage the application prospect of COMDES-II, in the sense that framework users are relieved from the error-prone manual coding effort, and as a result can focus on the high-level specification and analysis of the application domain models.

The design concepts provided by COMDES-II for modeling control software systems take both system architectural and behavioral aspects into account. On one hand, the hierarchical modeling approach provides for an open system architecture featuring succinctness and readability. On the other hand, the *separation-of-concerns* principle is extensively applied to help clearly specify various aspects of system behavior in the presented modeling context, which may facilitate the analysis of individual behavioral properties.

8. References

[1] T. A. Henzinger and Joseph Sifakis, "The Embedded Systems Design Challenge", *Proceedings of the 14th International Symposium on Formal Methods* (FM), Lecture Notes in Computer Science, Springer, 2006.

[2] E.A. Lee, "Embedded Software", *Advances in Computers*, Vol.56, Academic Press, London, 2002.

[3] J. Reekie and E. A. Lee, "Lightweight Component Models for Embedded Systems", *Technical Memorandum UCB/ERL M02/30*, University of California, Berkeley, CA 94720, USA, October 30, 2002.

[4] C. Angelov, Xu Ke and K. Sierszecki, "A Component-Based Framework for Distributed Control Systems", *Proc. of the 32nd Euromicro Conference on Software Engineering and Advanced Applications* (SEAA 2006), Cavtat-Dubrovnik, Croatia, August, 2006.

[5] Czarnecki, K. and Eisenecker, U. W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Professional. 1st edition, June, 2000.

[6] Xu Ke and K. Sierszecki, "Generative Programming for a Component-based Framework of Distributed Embedded Systems", *Proc. of the 6th OOPSLA Workshop on Domain Specific Modeling*, Portland, Oregon, USA, Oct. 2006.

[7] C. Angelov and J. Berthing, "Distributed Timed Multitasking: a Model of Computation for Hard Real-Time Distributed Systems", *Proc. of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems* DIPES'06, Braga, Portugal, Oct. 2006.

[8] J. Liu and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software", *IEEE Control Systems Magazine: Advances in Software Enabled Control*, Feb. 2003, pp. 65-75.

[9] A. Ghosal, T.A. Henzinger, C.M. Kirsch and M.A.A. Sanvido, "Event-Driven Programming with Logical Execution Times", Proc. of HSCC 2004, *Lecture Notes in Computer Science,* vol. 2993, pp. 357-371.