Modeling In Simulink Temporal Behavior of a Real-Time Control Application Specified in HTL

Daniel Iercan, Elza Cîrciu

"Politehnica" University of Timisoara, Department of Automation and Applied Informatics
300223 Timisoara, Bd. Vasile Parvan No.2, Romania
e-mail:{daniel.iercan, elza.circiu}@aut.upt.ro

Abstract: Development of a real-time control application usually starts with the design of the control algorithm using modeling tools such as Simulink. Once the control solution has been designed and validated through simulations it is implemented using one of the existing real-time programming technologies. Such a technology is Hierarchical Timing Language (HTL), which can specify timing and interactions between periodic tasks. Although the control solution is tested through simulations, the timing of the application that will implement the control solution in most of the cases it is not. This paper presents a method of modeling in Simulink temporal behavior of a real-time control application which is specified in HTL, thus allowing testing both functionality and temporal behavior at once.

Keywords: real-time, control application, Simulink, timing, model.

1. INTRODUCTION

In the process of development of real-time control application offline testing plays an important role. Simulink [1], which is a tool for modeling, simulating, and analyzing multi-domain dynamic systems, is widely used for designing control algorithms. Simulink offers the possibility to model both plant and controller dynamics, thus allowing testing of a control algorithm before it is used on the real plant. Nevertheless even if the model of the plant is very close to the real plant, simulating the control algorithm in Simulink it is not enough to ensure that the implementation of the control algorithm for a given platform will work. This is because when the control algorithm is simulated in Simulink it does not consider the timing of the application that will implement the controller, thus the implementation could introduce some unknown delays, which have not been considered in the design of the control algorithm, and which in the end could make the control application not to work properly.

Hierarchical Timing Language (HTL) [2] is a relatively new programming language that can be used to specify timing of a real-time application. HTL can not be used to specify functionally of a real-time application, which has to be implemented in a regular programming language (e.g.: C, Java, etc.). HTL has been developed as an extension of Giotto [3]. Development of a real-time control application using HTL consists of specifying the timing of the application as an HTL description and of implementing the functionality in a regular programming language. Currently there are HTL implementations for UNIX operating systems [4], for Java [5], and for Robostix [6] (a board based on a 16 MHz microcontroller).

In this paper it is presented a method for modeling HTL descriptions in Simulink. The HTL compiler [2, 4] has been extended in order to compile an HTL description into a Simulink model. Using this new feature of the HTL compiler, it is possible to simulate not only the control algorithm, but also the timing of the real-time application that will implement the controller, which should improve development of real-time control application using HTL. Beside of being able to simulate the control algorithm and the timing of the application that will implement the algorithm, modeling an HTL description in Simulink has another important advantage: the ability to generate C code for tasks directly from the Simulink schema, using Real-Time Workshop [7]. Thus, once the tasks have been modeled in Simulink functional C code for them can be generated automatically. Generated C code can then be used as tasks implementation.

The rest of the paper is structured as follows: in Section 2 we will compare the work presented in this paper with other similar work, in Section 3 there will be presented elements of HTL syntax, in Section 4 it is presented a case study that will be used to explain the modeling of an HTL description in Simulink, in Section 5 it is presented the modeling procedure, Section 6 presents the results of using the modeling procedure for the case study, finally Section 7 concludes the paper.

2. RELATED WORK

Modeling timing of a real-time application in Simulink is not a new idea; it has been done before for Giotto [8]. Nevertheless, modeling a Giotto program is different than modeling an HTL description, since Giotto has no hierarchical structure and no communicators. Generating code from a Simulink schema has been done for Giotto and for other languages. For Lustre it has been developed a tool chain [9] that can generate a Lustre program out of a Simulink model, which is the opposite of what it is presented in this paper for HTL, since in this paper from an HTL description it is generated the Simulink model, while for

Lustre the Simulink model is created first and then from it the Lustre program is generated.

3. ELEMENTS OF HTL SYNTAX

The main elements of an HTL description are represented by *communicators* and *tasks*. A communicator is a typed variable that can be accessed (read or written) only at particular moments in time. A communicator has a period associated with it, which identifies the moments in time when the communicator can be accessed. In other words a communicator represents a sequence of values. A communicator instance represents the value of a communicator at a moment when the communicator can be accessed. Communicator instances are defined relative to the period of the task that accesses the communicator. A task can access a communicator only if the period of the task is a multiple of the period of the communicator.

A task is a block of sequential code that contains no synchronization points. A task has a set of input ports and a set of output ports; the only way to communicate with the environment and with other tasks is through the input and output ports. The task model of execution used in HTL is the so-called Logical Execution Time (LET) [3]. Tasks in HTL can be either abstract or concrete. Abstract task are not executed at runtime they are only used as placeholders for concrete tasks. Concrete tasks on the other hand are executed at runtime.

Tasks can be grouped in a so-called *mode*. All the tasks that are invoked in the same mode will be executed with the same frequency. Modes offer support for sequential composition, namely, only one mode can be active at a particular moment in time, i.e., only the tasks invoked in that mode will be executed. The active mode can switch to another mode, i.e., tasks that are invoked in the mode that switches will be replaced by tasks that are in the destination mode.

Modes are group in a so-called *module*. Modules offer support for parallel composition. All the active modes in a set of modules are executed in parallel.

One or more modules form a *program*. Programs can be used in HTL to create a hierarchical structure. An HTL description can contain one or many programs, but there is only one root program. The root program specifies a generic temporal behavior, which is further refined by the rest of the programs in an HTL description.

The main advantage of the hierarchical structure is the possibility to simplify static checks such as schedulability check. Thus for an HTL description it is sufficient to check that the root program is schedulable for a given platform in order to say that the entire application is schedulable. This is ensured by the refinement constraints that have to be met by all the refining programs.

For a formal definition of the HTL semantics and for a definition of all the constraints that have to be met by an HTL description, please refer to [2].

As in the case of its predecessor, Giotto, HTL descriptions are not compiled directly into machine code but into so-called HE~code which is interpreted by a virtual machine, *Hierarchical Embedded Machine* (HE machine) [4]. The HE machine is an extension of the original E machine that has been developed for Giotto [10].

4.CASE STUDY

The method of modeling an HTL description into a Simulink schema will be presented by considering the HTL description that specifies the temporal behavior for the real-time control application for the Three Tanks System (3TS) plant.

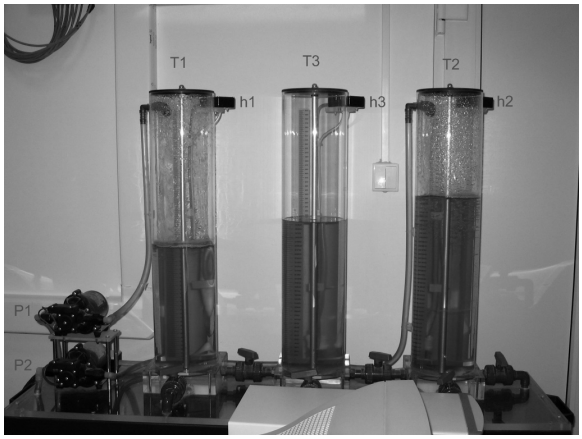

Fig.1. Three Tanks System Plant.

The 3TS plant (Fig.1) consists of three interconnected tanks, e.g., T_1, T_2, and T_3. Each tank is connected to a tap that can be used to drain water out of it; tank T_2 is connected to two such taps. Tank T_1 communicates to tank T_3 through an interconnection tap; tank T_2 communicates to tank T_3 through another interconnection tap. The plant also contains two pumps, e.g., P_1 and P_2, which are used to feed water into the system; P_1 is connected to tank T_1 and P_2 is connected to tank T_2. There are also three sensors (e.g.: h_1, h_2, and h_3) connected to tanks T1, T2, and T3, respectively that give information about the level of the water in each of the three tanks.

The objective of a controller for the 3TS plant is to control the water in tanks T_1 and T_2 to a prescribed target level, by adjusting accordingly the water that is fed into the system by the two pumps, namely, P_1 and P_2. Given that the 3TS plant is highly non-linear, the use of a simple control strategy is not enough to control the plant in any scenario. A control solution that is able to control the plant in any scenario has been presented in [10], the solution consists of using a control strategy for both tanks T_1 and T_2 that switches between a proportional (P) and a proportional-integral (PI) controller.

Fig. 2 depicts in visual syntax the structure of the HTL description that specifies temporal behavior of the application that implements the 3TS controller. The application consists of running in parallel the controller for tank T_1 and the controller for tank T_2, thus the root program contains three modules one for each tank controller (*T1* and *T2*), while the

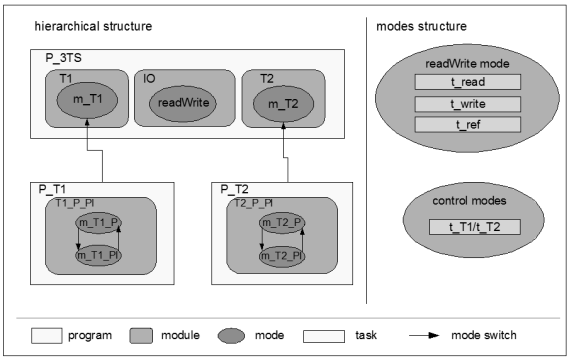

Fig.2. Graphical representation of the HTL description that specifies timing for the 3TS control application.

third module specifies the timing of the communication with the plant (*IO*). The two modules that specify timing for T_1 and T_2 controllers, respectively, are very similar; both modules contain a single mode (*m_T1* and *m_T2*, respectively), which invokes one abstract task, i.e., the control task (*t_T1* and *t_T2*). The two control abstract tasks, namely, *t_T1* and *t_T2*, are refined by programs *P_T1* and *P_T2*, respectively, in two concrete tasks: a P control task and a PI control task. The communication module contains one mode (*readWrite*), which invokes three tasks: *t_read* – reads the level of the water in tanks T_1 and T_2; *t_write* – sends the commands to the pumps; *t_ref* – reads the target values. All the modes in the description have a period of 500ms.

The flow of data between tasks in the HTL description is presented in Fig. 3. After task *t_read* reads the sensors values it writes them to communicators *h1* and *h2*, it also estimates if there is perturbation in tanks T_1 and T_2 and writes the results into communicators *v1* and *v2*. Task *t_ref* reads the target values for each of the tanks and writes them into the communicators *h1_ref* and *h2_ref*. Task *t_T1*(*t_T2*) reads communicators *h1*(*h2*) and *h1_ref*(*h2_ref*) computes the control law and writes the new command for pump $P_1(P_2)$ into communicator *u1*(*u2*). Communicators *u1* and *u2* are read by task *t_write*, which sends the commands to the pumps.

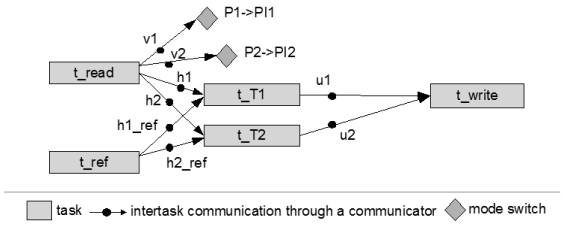

Fig.3. Data-flow between tasks in the 3TS control application.

5. MODELLING HTL PROGRAMMING CONSTRUCTS IN SIMULINK

A control application monitors sensor values of a plant based on which it will compute a command, which will be sent to the plant, throughout the actuators, in order to ensure that the plant will achieve a desired state, which is defined through the target value.

An HTL description that is to be translated into a Simulink model consists of one and only one communication module with the plant and with the environment, which does the sensing and actuating, and which reads the target values. Otherwise the program can contain as many modes, modules, and programs as they are needed, but they will communicate with the environment and with the plant only through the communication module.

Bearing in mind the idea of a control application, the top level of a Simulink model generated from an HTL description will have two subsystems that communicate to each other. One subsystem represents the HTL description (i.e., the control application), this subsystem will be referred to as the *controller subsystem*, while the other subsystem will represent the plant, and it will be referred to as the *plant subsystem*. The controller subsystem reads the sensor values from the plant subsystem and the target values, and writes the commands back to the plant subsystem. In Fig. 4. it is presented the first level of the model that has been generated for the *3TS* aplication.

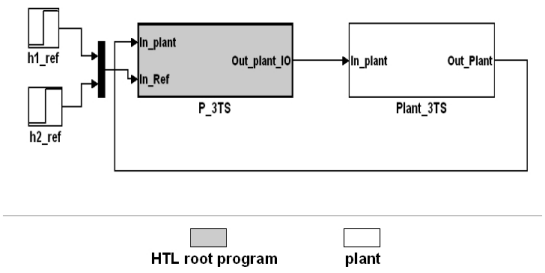

Fig 4. 3TS model - first level

For any HTL program an atomic subsystem block will be generated. The block will contain an atomic subsystem block for each HTL module in the HTL program and a subsystem block for each communicator. The difference between the root HTL program and any child HTL program consists in the fact that for the root program a *digital clock* is generated, while a child program receives the clock from the parent program. Also the inputs and outputs of the root program communicate with the plant, while for a child HTL program they are connected to communicators in the parent program. The digital clock that is generated for the root program is used to simulate time events and it should have a period that is at least an order of magnitude smaller than the smallest period in the HTL description. In Fig. 5 it is presented the content of the atomic subsystem that represents the root program of the 3TS HTL description. An HTL communicator is modeled in Simulink as a subsystem that contains a merge block connected to a memory block. The subsystem always has one input, but it can have as many inputs as modules, since for each module that writes to a certain instance of the communicator there has to be an input. At most one input can carry a value at a particular moment in time; this is ensured by HTL constraints since no communicator instance can be written from two different modes. Fig. 6 presents the model of the *h1* communicator from the 3TS HTL description.

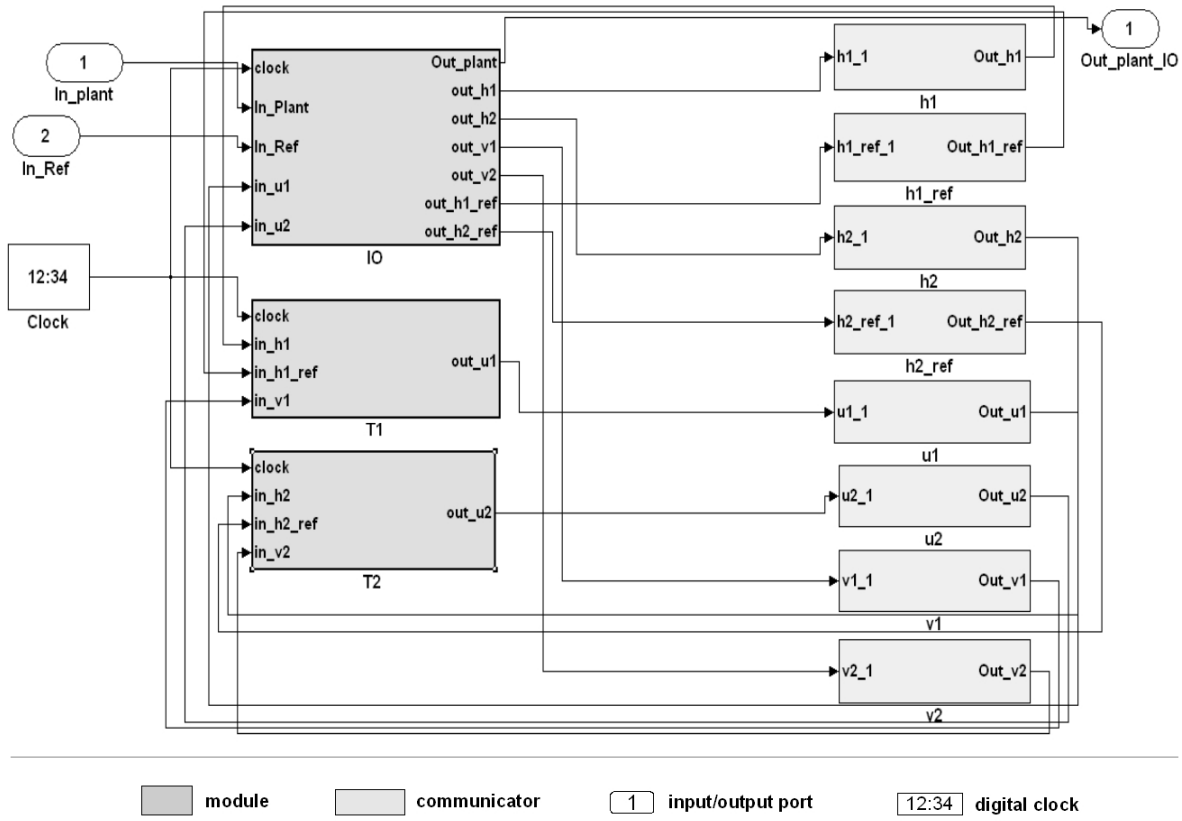

Fig 5. Simulink model for 3TS root program

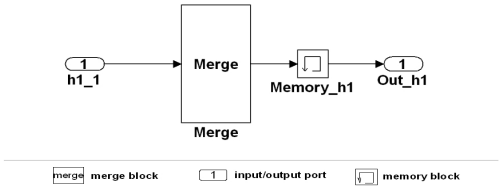

Fig. 6. Simulink model for communicator *h1*.

The Simulink model of a module consists of a set of action subsystems, which represent the modes that belong to the module, a mode selector, which determines the mode that should be invoked (active mode), and for each communicator that is written in more than one mode there has to be a merge block in order to merge together the signals from different modes and to produce only one signal that will be connected to the input of the communicator. The Simulink representation of the module receives as an input the clock signal from the containing program, which is broadcasted to each Simulink model of each mode in the module. Besides the clock input, the Simulink model of a module also has as inputs all the communicators that are read in the modes belonging to the module. The outputs of the Simulink representation of a module are represented by all the communicators written in any of the modes in the module. In order to implement mode switching each mode is associated with a unique integer number and each mode generates a signal that specifies which is the next mode. The mode selector block reads the next mode signals from each mode and activates the action subsystem that corresponds to the mode that is to be executed. For each mode in a module, the mode selector block has an output, which is connected to the action port of the subsystem which models the mode. Also for modes that can switch to other modes, the mode selector block generates a reset signal, which ensures that a mode does not switch in the first period when the mode is executed. Fig. 7 presents the Simulink model of the T1_P_PI module.

A mode selector consists of a merge block that merges together the next mode signals from all the modes in the containing module, a memory block which stores the last enabled mode (this block is needed in order to avoid algebraic loops), and a switch case block which does the mode selection based on the signal that comes from the memory block. Beside the role of selecting the active mode, the mode selector block has a second role, which is the generation of reset signals. The reset signals are generated for all modes that can switch to any other mode; thus for each such mode there will be a reset mode generator. The reset signal is active only in the first period a mode is active; in order to achieve this, the value of the merged next mode signal at the beginning of the active mode period is stored until next mode period, the stored value is then compared to the current value of the same signal and if they are different, then one of the reset signal is activated based on the value of the merged next mode signal (which actually specifies which mode will be active); this will ensure that the reset signal for

a mode is active only in the first period when the mode was activated. The mode selectors that belong to modules from other programs than the root program have a third role: to reset the active mode to the default mode whenever one of the parent modes switches. In order to do this a reset signal generated by the parent mode that switches is used in order to force the current active mode to the start mode of the module. In Fig. 8 it is presented the mode selector for the T_P_PI module.

The Simulink model of a mode consists of a set of blocks that implement mode clock generator, a set of blocks that implement tasks invoked in the mode, and a set of blocks that implement mode switches. Instead of task invocations, the Simulink representation of a mode can contain a subsystem representing the program that is refining the current mode. This is the case of mode m_T1 that will be refined by program P_T1 as it can be seen in Fig. 9. The Simulink model for module T1_P_PI presented above belongs actually to the subsystem representing program P_T1.

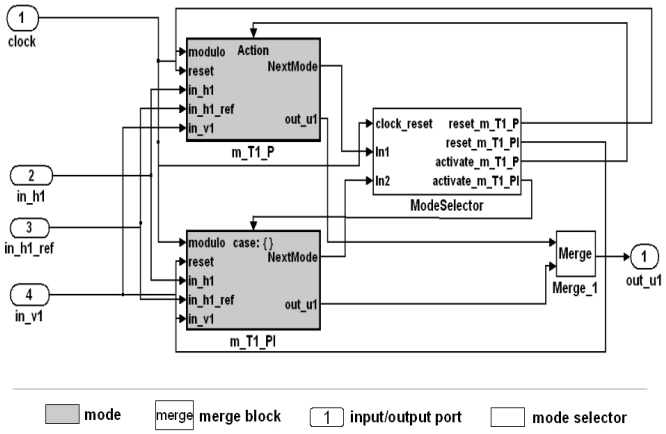

Fig. 7. Simulink model for module *T1_P_PI.*

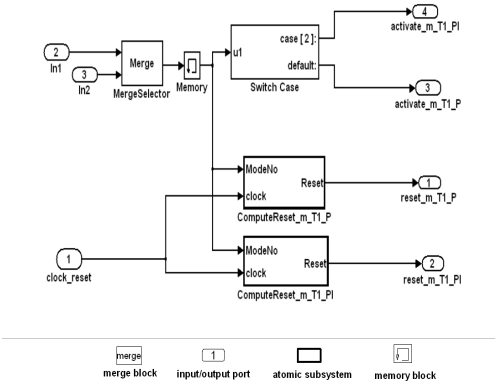

Fig. 8. Simulink model for mode selector for T_P_PI module

The mode clock generator computes modulo function between the system clock scaled with 10000 and the period of the mode expressed in milliseconds. The mode clock has to be scaled by 10000 in order to achieve high precision.

For modes that are not in the root program, the mode clock is generated by the parent mode (i.e., the parent mode and the child mode have the same period in a well-formed HTL program). For each task in a mode there is an atomic subsystem that reads the mode clock signal and the communicators read by the task; the task block has an output port for each output port of the task being modeled. If two task invocations write to two different instances of the same communicator, then the two outputs will be merged by a merge block. For each mode switch in a mode a trigger subsystem, which implements the switch condition, is generated. The switch block has an input for each communicator read by a mode switch and one single output, which can take two values: 0 when the switch is not enabled and 1 when the switch is enabled. The switch block executes only at the beginning of the period; this is ensured by a block that reads the mode clock and if the clock is between 0 and 10, then the switch block execution will be triggered. The output of each switch block is read by an action subsystem which computes the next mode based on all the switch blocks outputs. If a mode contains at least one switch, then it also has to read a reset signal, which is active in the first period the mode is executed and which is used to disable mode

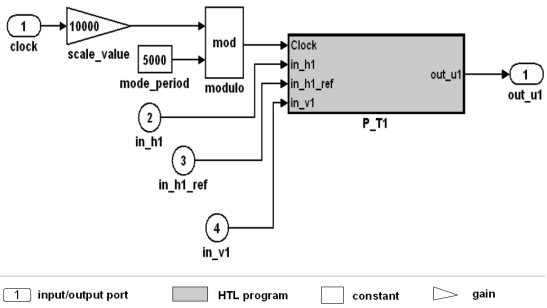

Fig. 9. Simulink model for mode m_T1

switch logic and to force the next mode signal to the integer value that is associated with he mode modeled by the block. Fig. 10 presents the model for the mode m_T1_P in program P_T1. An HTL task invocation is modeled as an atomic subsystem, which contains a triggered subsystem for each task input and output; the implementation of the task functionality is also done in a triggered subsystem. Logic

based on the mode clock and on the communicator instance that is read/written is generated to activate the corresponding triggered subsystem. For the activation of the inputs a delay block is used in order to allow a communicator to be written before it is read. The delay is very small as compared to the entire period of a mode, thus the timing is not affected. The triggered subsystem that represents the task functionality has to be activated when the latest communicator is read, for tasks with dependencies activation of the triggered subsystem that implements the functionality has to consider the latest communicator read by any task in the dependency list. In Fig. 12 it is presented the Simulink model for the invocation of task t_T1_P, which reads instance four of communicators h1 and h1_ref and writes back to instance five of communicator u1. An HTL mode switch is modeled as a triggered subsystem, which is activated at the beginning of the period of a mode. A mode switch block consists of an *if* block which implements the switch condition and which activates one of two blocks. If the condition is true, then a block that has always value "1" at output is activated, otherwise a block that has always value "0" at output is activated. The outputs of the two blocks are merged and written to a single output. In Fig. 11 it is presented the Simulink model of the mode switch that switches from mode m_T1_P to mode m_T1_PI.

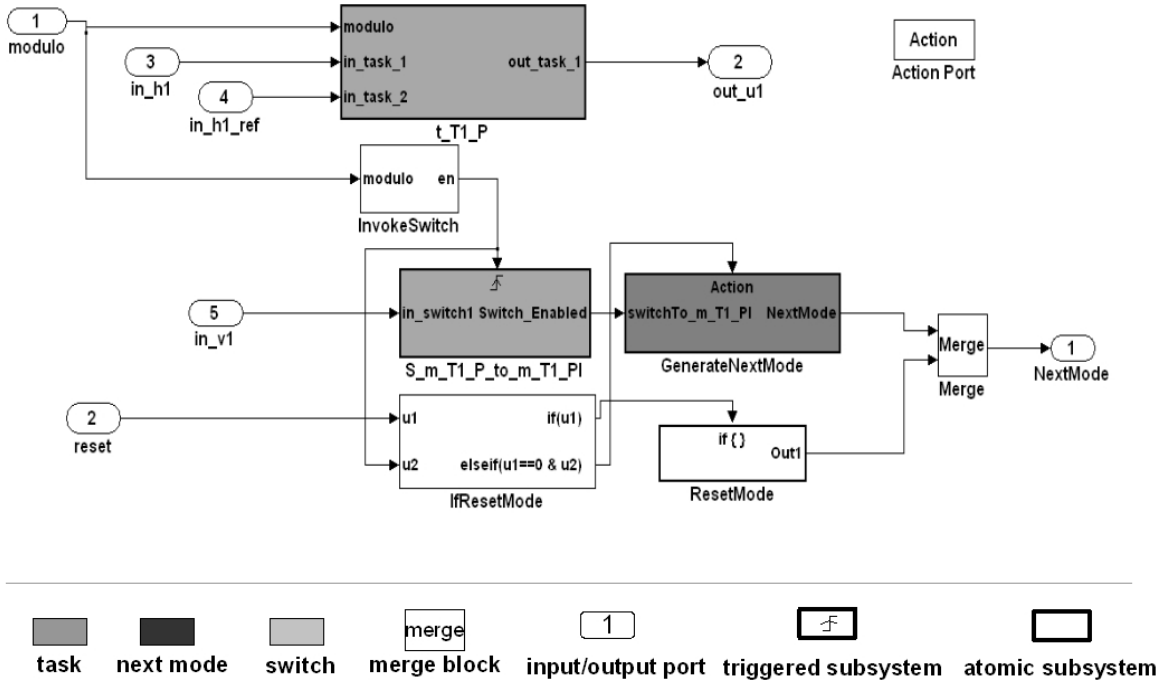

Fig. 10. Simulink model for mode m_T1_P

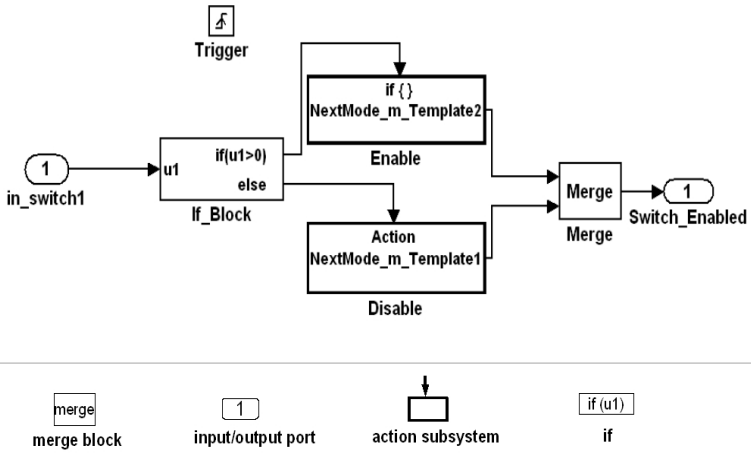

Fig. 11. Simulink model for mode switch m_T1_P to m_T1_PI

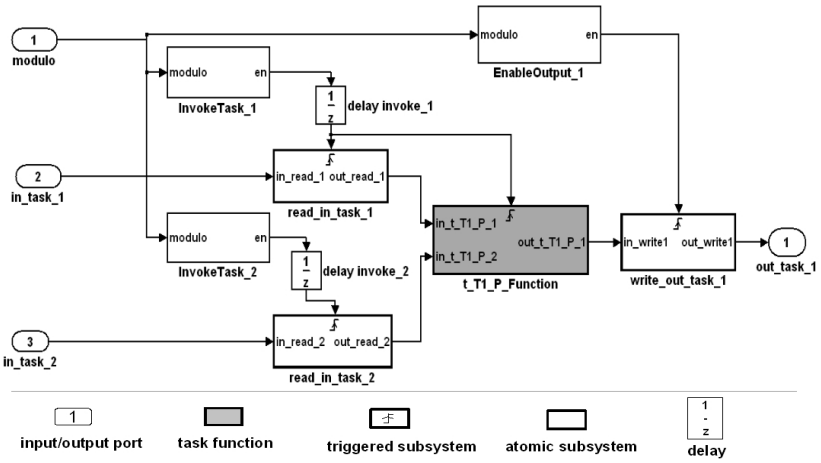

Fig. 12. Simulink model for task t_T1_P

6.RESULTS

In order to validate and test the modeling method presented above we have implemented a real-time control application for the 3TS plant using this modeling method. We have used the generated Simulink model to implement the functionality of all the tasks in the application and we have used the Real-Time Workshop Embedded Coder to generate functional code directly from the Simulink model. We have simulated the Simulink model that models both the timing and the functionality of the application. In Fig.13 it is presented the evolution of the level of water in tank T_1 when simulating the Simulink model. After we have obtained the HTL real-time control application by compiling together timing of the application represented by the HE code program and functionality generated directly from the Simulink model, we have used the application to control a simulated version of the 3TS plant. In Fig. 14 it is presented the evolution of the water in tank T_1 when using the HTL control application.

In both scenarios we have started without any perturbation in tank T_1 (no tap was open), then at moment *t=200s* we have opened the evacuation tap. In Fig. 13 and Fig. 14 it can be seen that in both cases the control time is the same and that both controllers react the same way to the perturbation. Thus we can say that the Simulink model of the HTL real-time control application is accurate enough.

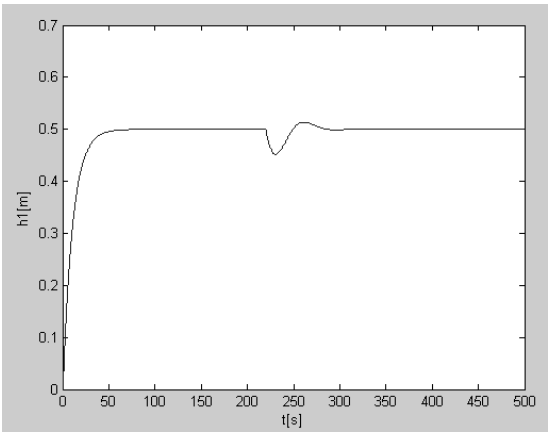

Fig. 13. Water level in tank T_1 when simulating in Simulink.

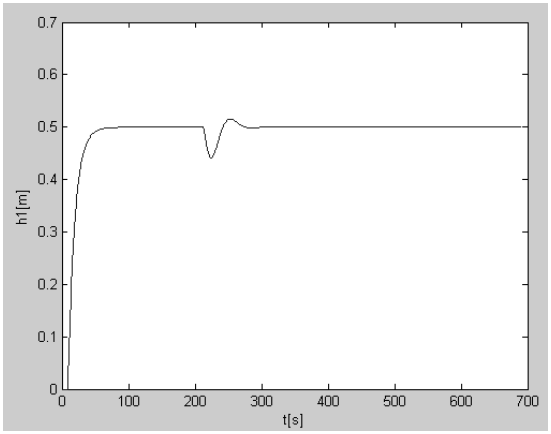

Fig. 14. Water level in tank T_1 when using the HTL control application.

7. CONCLUSION

We have presented a method of modeling an HTL description into a Simulink model. We have extended the HTL compiler so that it can compile an HTL description into a Simulink model. We have tested the method by implementing a real-time control application for the 3TS plant. The results obtained by simulating the Simulink model and those obtained by running the control application are very similar, thus we can conclude that the modeling method is accurate.

Using this modeling method, a new methodology for developing real-time control applications with HTL can be defined. The methodology consists of the following steps. Based on the initial specifications for the control application, the timing of the application can be extracted and implemented as an HTL description. From the HTL description using the HTL compiler it can be generated both the HE~code (represented as C code) and a Simulink model. The Simulink model of the HTL description will be used to develop the control algorithm, and it can also be used to simulate and test timing of the application; in case of errors the HTL description can be reviewed. After the control solution has been developed and tested in Simulink, the Real-Time Embedded Coder can be used to generate C code for those blocks that implement tasks, which represent the

functionality of the application. In the last step the HE~code and the functionality are compiled together with the C implementation of the E~machine and it results the real-time control application which can be tested on the real plant. If this test fails then either the functionality or the timing of the application can be reviewed. The process is repeated until a stable application results.

REFERENCES

Auerbach, J., Bacon, D.F., Iercan, D., Kirsch, C.M., Rajan, V.T., Rock, H., and Trummer, R., Java takes flight: Time-portable real-time programming with exotasks. In Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools, San Diego, California, USA, **51--62**, 2007.

D. Iercan, and M. Mezin, A Distributed Multimode Real-Time Controller for the Three Tanks System. ReNeCoSy'2008, Proc. Of the 8th International Conference On Tehnical Informatics - CONTI'08, ISSN: 1844-539X, Vol. 3, pag. 67-70, 5-6 June, 2008.

Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C.M., and Sangiovanni-Vincentelli, A.L., A Hierarchical Coordination Language for Interacting Real-Time Tasks. In Proc. EMSOFT, Seoul, South Korea, **134--141**, 2006.

Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C.M., and Sangiovanni-Vincentelli, A.L., Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code, In Workshop Proc. APGES, Salzburg, Austria, 2007.

Henzinger, T.A., and Kirsch, C.M., The Embedded Machine: Predictable, Portable Real-Time Code. ACM TOPLAS, 29(6), **33--61**, 2007.

Henzinger, T.A., Horowitz, B., and Kirsch, C.M., Giotto: A Time-triggered Language for Embedded Programming. Proceedings of the IEEE, 91(1), **84--99**, 2003.

Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., and Pree, W.. From Control Models to Real-Time Code using Giotto. IEEE Control Systems Magazine, 23(1), **50--64**, 2003.

Real-Time Workshop. http://www.mathworks.com/products/rtw/.

Robostix.http://docwiki.gumstix.org/index.php/Robostix.

Simulink. http://www.mathworks.com/ products/ simulink/.

Tripakis, S., Sofronis, C., Caspi, P., and Curic, A., Translating Discrete-Time Simulink to Lustre. ACM Transactions on Embedded Computing Systems, 4(4),**779--818**, 2005.