Validating UML models of Embedded Systems by Coupling Tools*

Jozef Hooman[1], Nataliya Mulyar[2], Ladislau Posta[2]

[1]Embedded Systems Institute, Eindhoven & University of Nijmegen, the Netherlands
[2]Eindhoven University of Technology, the Netherlands

Abstract

To support multi-disciplinary development of embedded systems, a coupling has been realized between a UML-based CASE tool (Rose RealTime) - to model the embedded software - and a tool for the modeling of the continuous dynamics of physical parts of the system (Simulink). The aim is to allow simultaneous simulation of the models in both tools, thus allowing an early exploration of the possible design choices over multiple disciplines. A first prototype of the coupling has been implemented, with an emphasis on realizing a common notion of time and a proper treatment of timers and data exchange.

1. INTRODUCTION

The general aim of this work is the improvement of multi-disciplinary development of embedded systems. The focus of this paper is on software-controlled machines where especially the relation between mechanical engineering and software engineering is important. Note, however, that there are also relations with other disciplines, such as electrical engineering. The work described here is part of a collaboration with the company Océ, a producer of high-volume printers and copiers, but the proposed solutions could also be applicable in other domains such as avionics and automotive.

Although the different disciplines are tightly coupled in the considered embedded systems, their development is often a rather sequential, mono-disciplinary, process. Typically, first the mechanical part is designed, next the hardware infrastructure is fixed, and finally the embedded software is developed. This approach can create large problems, especially for the software engineers. For instance, choices about the placements of sensors (and implicitly the occurrence of interrupts), control rates, control delays, hardware, etc., have a strong influence on the complexity of the software. Moreover, usually many implicit assumptions are made, which first become visible at system integration. This easily leads to non-optimal solutions.

Within each discipline, a common solution is the frequent use of models to detect problems as early as possible. For instance, in the software domain a lot of effort is put on model driven development, based on UML models. Moreover, mono-disciplinary modeling is usually supported by tools that allow some form of execution or simulation. Lacking, however, is the possibility to combine tools of different disciplines, to

* This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program. The first author is partially supported by IST-2002-33522 project OMEGA.

investigate the mutual influence of modeling choices. Our aim is to couple currently used tools to allow simultaneous simulation of the models from different disciplines. (Related work is described at the end of this section.)

Given the collaboration with Océ, we have implemented a coupling between the UML-based CASE tool Rose RealTime (currently renamed to Rose Technical Developer) of IBM Rational [8] and Matlab/Simulink of The Mathworks [10]. Rose RealTime (Rose-RT) supports the ROOM methodology [9] for the development of software for real-time reactive systems. It is used at Océ to define a re-usable software architecture, which is instantiated for a particular printing/copying machine. Next the tool allows the generation of code for a particular target platform, thus obtaining a direct connection between a model and the generated code. Simulink is used at Océ to model the mechanical layout of the machine and to experiment with, for instance, the shape and the length of the paper path, the placement of motors and sensors, and the paper speed.

As an example, our coupling allows the combination of a continuous-time model of a physical dynamical system in Simulink, e.g. representing the dynamics of a particular machine, with a discrete-time control algorithm in Rose-RT, as depicted in Figure 1.

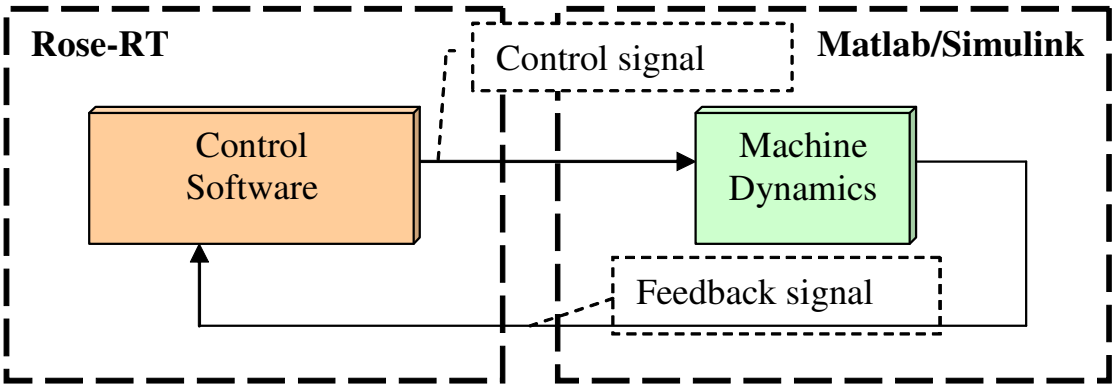

Figure 1 Combined Models

By establishing a proper notion of simultaneous simulation of these models, one can quickly investigate the effect of changes in the control strategy, the software execution times, or the effect of different motor characteristics. This also allows a comparison with a model where part of the control (e.g. low-level motor control) is modeled in the Matlab environment (e.g. using TrueTime [13]) and a supervisory control part in Rose-RT, as shown in Figure 2.

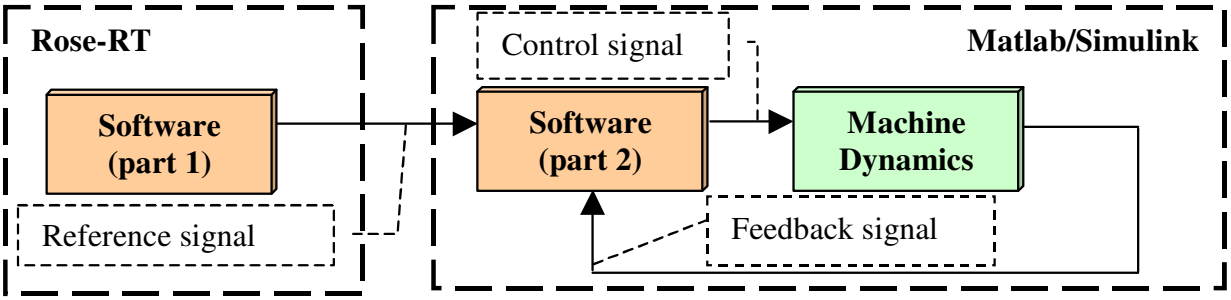

Figure 2 Distributed Software Control

In fact, in the above example, TrueTime could have been used also to represent all discrete real-time control, but we have chosen a UML-based CASE tool, since UML is already an ad hoc standard in software engineering and it allows a coupling with large object-oriented software architectures that include not only

control but also other aspects such as error handling and diagnosis. Moreover, a tool such as Rose-RT enables code generation for different platforms.

Realizing the desired tool coupling is far from trivial. The two main challenges are:

- *Conceptual correctness*. The coupling should be such that the simultaneous simulation of models in both tools gives meaningful results. In particular, this means that there must be a common notion of simulation time in combination with a proper exchange of data and messages. For instance, control data computed by the UML model is to be used in the Simulink model at the right moment in time, taking the duration of the computation into account. Moreover, time-outs generated by timers in the Rose-RT model should correspond to the simulated time in Simulink.
- *Technical implementation*. The coupling software should be properly designed to allow, for instance, a change to another UML tool without too much effort. Moreover, suitable ways should be found to allow the tools to communicate and to run a simulation mode simultaneously. The current version of the coupling works in a Windows environment with Matlab release 13, Simulink version 5.0, and Rose-RT version 6.5.341.0.

Such a coupling between Rose-RT (or similar real-time UML tools) and Simulink models has not been described before. Related is the work on the High Level Architecture (HLA) [3], a general-purpose architecture for the coupling of simulation tools. However, HLA cannot be used for our purpose, because UML-based CASE tools, such as Rational Rose RealTime do not fit into the HLA framework since they do not have the required simulation mode with a well-defined notion of simulation time.

An alternative solution to the modeling of mixed discrete-continuous systems has been followed in the HyROOM approach [12], where the ROOM/Rose-RT notation has been extended with continuous elements. The main parts of the resulting tool have been mapped into HyCharts [5], a formal framework for hybrid systems. Along this line, there are several formal approaches that allow checking properties of hybrid systems (modeling both discrete and continuous aspects), such as HyTech [6] and Checkmate [2]. Another approach is the use of generic modeling environments, such as Ptolemy [7] and the Generic Modeling Environment (GME) [4], that allow the development of domain specific modeling tools. Our aim, however, is to allow engineers to continue working with their well-known mono-disciplinary tools, without having to redo their modeling work in some multi-disciplinary modeling environment.

The rest of this paper is structured as follows. Sections 2 and 3 contain a very brief presentation of the tools used (Rose-RT and Simulink, respectively), describing them only as far as needed to understand the coupling. The main concepts of the coupling are explained in Section 4. Details of the implementation are given in Section 5. Concluding remarks can be found in Section 6.

2. ROSE REALTIME

Rose-RT is a UML-based CASE tool for the development of complex reactive software. Typically, a UML model in Rose-RT consists of a number of active objects, also called *capsules*, which communicate by sending and receiving messages via *ports*. Messages may have different priorities. A port must refer to a *protocol*, which represents a set of messages that can be exchanged between capsules. The behavior of a capsule is modeled by means of a hierarchical state diagram. Transitions in a state diagram are triggered by the receipt of messages or time-outs. Actions on a transition may change local variables, send messages, or set timers.

Given a complete model, the Rose-tool can generate code for a specific target platform, using the characteristics of that platform. Model execution is based on the Service Layer which provides general services, e.g. it contains controllers, which are responsible for queuing and delivering messages among capsules, and timing services that can be used in the generated code.

The implementation of the Service Layer depends on the target platform on which the program should run. Hence, at the code generation and compilation step for a Rose-RT model, the toolset links the user-defined code with a services library for the particular platform on which the model is intended to run.

The Service Library also contains services for concurrency control and thread management. Capsules can belong to different logical threads. Logical threads are mapped to a set of concurrent physical threads defined by the developer of the Rose-RT model. No other capsules in a thread can execute until the currently executing capsule returns control to the main loop of that thread. However, other capsules on other physical threads may be executing concurrently.

Each thread has a separate message queue and its own controller object that is responsible for queuing and delivering messages among capsules in this thread. This controller object contains the basic message delivery and processing loop. The underlying operating system is responsible for switching control among active physical threads. The operating system may preempt one physical thread in the middle of execution to switch to another physical thread. Each thread can be assigned a separate priority, so that the designer has some control over the scheduling.

Message processing, which is provided by the Services Library, is based on the following steps. During start-up, the initialization message is the message with the highest priority. When a capsule processes the initialization message, the capsule's initial transition segment is executed. During the main processing loop the controller object takes the next highest priority message from the message queues and delivers it to the receiver capsule and invokes that capsule's behavior to process the message. Each capsule processes the current message to the completion of the transition chain. This is referred to as *run-to-completion* semantics. When the capsule has completed processing a message, it returns control to the controller. The controller continues this loop until there are no more messages to be processed.

One *step* in Rose-RT is associated with processing the next message of the highest available priority. A step terminates when all actions associated with the respective message are performed.

There are two ways of testing the generated and compiled code. The first way is to run the executable on the intended platform. The second way is to execute the model step-by-step on a simulated platform, but then correct timing is not guaranteed and only the reactive response to messages can be tested.

The Timing Service of the Rose-RT Services Library provides the model developers with general-purpose timing facilities based on both absolute and relative time. There are two types of timers: a *one-shot timer* and a *periodic timer*. The *one-shot timer* expires only once, after the specified time duration (relative time), or at a specified time (absolute time). The *periodic timer* is set to timeout repeatedly after the specified duration until the timer is explicitly cancelled. It does not need to be reset after expiration. The use of the periodic timing service will provide more accurate timing than repeatedly resetting a *one-shot timer*.
The precision of the Timing Service depends on the granularity of the timing supported by the underlying operating system. Note that the granularity of the timing supported on most real-time operating systems is much finer than that of general-purpose workstation operating systems, such as UNIX and Windows-NT. The Timing Service does not guarantee absolute accuracy, since it is platform dependent. This means that intervals between timer creation and timer expiration can take slightly longer than specified, and events scheduled for a particular time may in fact happen slightly after the actual time has occurred.

3. MATLAB/SIMULINK

This section contains a short description of Simulink of The MathWorks, mainly based on the tool documentation [11]. A Simulink model is represented graphically by means of a number of interconnected *blocks*. Lines between blocks connect block outputs to block inputs. Blocks may have states, which may consist of a discrete-time and a continuous-time part.

The output of a block is computed by an *output function*, based on its input and its current state and time. Similarly, an *update function* calculates the next discrete state. A *derivative function* relates the derivatives of the continuous part of the state to time and the current values of the inputs and the state.

During the simulation of a Simulink model, the outputs, inputs and states are computed at certain intervals, from a start time to an end time, as specified by the user. The successive states of a system are computed by a so-called *solver*, a Simulink-specific program. Since no solver is suitable for all models, there are several types of solvers. The solvers use numerical integration to compute the continuous states of a system from the state derivatives specified by the model. Each solver uses a different integration method, allowing the selection of the most suitable method for a particular model.

The successive time points at which the states and outputs are computed are called *time steps***.** The length of time between steps is called *step size*. The step size depends on the type of the solver used, the characteristics of the Simulink model, and the existence of discontinuities of the continuous states (Simulink checks for such discontinuities – this is called zero crossing detection – and if it detects one within the current step, the precise time at which zero crossing occurs is determined and additional time steps are taken).

There are several types of solvers. *Fixed-step solvers* use a fixed step size. *Variable-step solvers* change the step size during simulation. They reduce the step size to increase accuracy when states are changing rapidly and increasing the step size to avoid taking unnecessary steps when states are changing slowly. This requires some additional computation each step, to determine the step size, but can reduce the total number of steps and hence the duration of the simulation. For purely discrete models there are *discrete solvers*. *Continuous solvers* compute continuous states using numerical integration. Simulink provides an extensive set of fixed-step and variable-step continuous solvers, each implementing a specific numerical integration technique for solving the ordinary differential equations that represent the continuous states of dynamic systems.

The solvers monitor the error at each time step; they compute the local error, which is the estimated error of the computed state values. If the local error is greater than the acceptable error for any state, the solver reduces the step size and tries again.

Simulation of a Simulink model starts with the initialization phase, where e.g. library blocks are incorporated, block parameters are evaluated, memory is allocated and the execution order of the blocks is determined. Next, Simulink enters a simulation loop, consisting of *simulation steps*. During each simulation step, Simulink executes all blocks of the model in the order determined during initialization. This execution order does not change during the simulation. For each block, Simulink calls functions that compute the block's states, derivatives, and outputs for the current sample time. This continues until the simulation is complete.

4. Main Concepts of the Coupling

First, the main decisions taken to establish a correct coupling are presented, namely, the notion of time in Section 4.1 and the global coupling architecture in Section 4.2. Next, Section 4.3 describes the initialization of the simulation. Section 4.4 contains the conceptual architecture. More details can be found in Section 5.

4.1 Notion of time

The most important decision concerns the notion of time to be used for the simulation. Observe that the timing of Rose-RT is strongly coupled to the timing service of the operating system of the target system on which the model is running. Moreover, timing is not respected in the step-by-step simulation. Hence, we concluded that the timing of Rose-RT is not suitable for our purpose and decided to use the notion of simulated time of Simulink instead. The alternative is to use a separate, independent, notion of time, but this would also require new implementations of solvers, redoing a lot of things already available in Simulink.

To be able to establish a proper notion of simulation time, which faithfully reflects the execution of both models, somehow the execution time of the transitions in the Rose-RT model has to be taken into account. We assume that this information is available, representing an assumption on the underlying platform.

4.2 Global coupling architecture

Another decision to be taken is the global architecture of the coupling. A possible approach, depicted in Figure 3, is to extend each tool with a specific component, which communicates directly with the other tool.

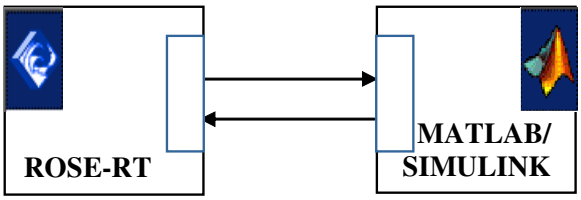

Figure 3 Tightly Coupled Tools

Instead of such a tight coupling, we decided to use a more loosely coupled architecture by introducing a third component called *Multidisciplinary Coupling Tool* (*MCT*), as shown in Figure 4. Observe that each tool contains an add-in, which is responsible for the communication with the MCT component.

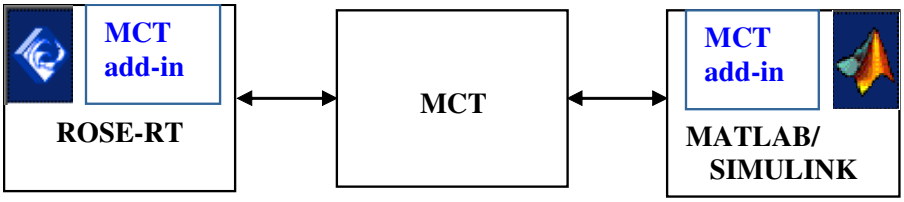

Figure 4 Loosely Coupled Architecture

By introducing such an MCT interface, the modeling tools do not need to know about each other and it becomes much easier to change, for instance, switching to another UML-based CASE tool. Moreover, it makes it easier for the engineers to establish a coupling without knowing much about the details of the models of the other discipline.

To obtain proper timing of the UML models, we have redefined the timing service of Rose-RT such that it gets the current notion of time from the MCT component, which passes on the notion of time it receives from Simulink.

4.3 Initialization

The Simulink model is extended with blocks that obtain data and timing info from the Rose-RT model via the MCT and that ensure that the data is used at the appropriate moment of time. Also the setting of a timer in the UML model is communicated to these added Simulink blocks, which then generate the time-out.

To run a simulation, we have to initialize the simulation environment as follows:

1. Start Rose-RT and open a UML model which has been prepared for coupling
2. Start Matlab/Simulink and open a Simulink model which has been prepared for coupling
3. Start the simulation in the Simulink model, which:
    a. Starts the Simulink timer
    b. Sets the time of MCT to the current Simulink simulation time
    c. Starts the Rose-RT Target Run-Time System; the Rose-RT environment then requests the system clock, which by our modifications means that it gets the time of the MCT, next the Rose-RT model waits for external triggers

4.4 Conceptual View

The conceptual architecture presented in Figure 5 contains three components: *Rose-RT*, *MCT*, and *Matlab/Simulink*. Each of these components consists of several modules, as explained below.

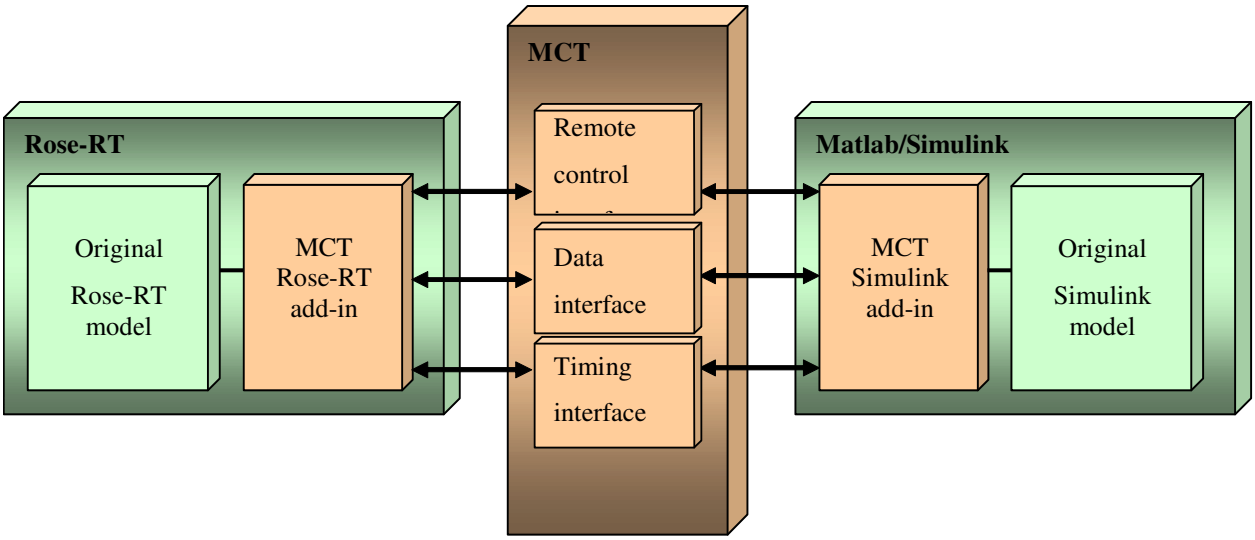

Figure 5 Conceptual Architecture View

The *Original Rose-RT model* and the *Original Simulink model* depict the original models supplied for coupling. The *MCT Rose-RT add-in* is a set of UML-model entities supplied with the MCT, which must be added to the original Rose-RT model; the add-in enables communication to the Rose-RT model via ports. In general, a Rose-RT model may communicate with external applications through external ports only. An external port can only identify the presence of a Rose-RT-specific type signal. This actually means that data cannot be send to an external port of a capsule. However, we can associate an external port with a signal, which notifies that the data supplied by Matlab/Simulink model is available in some data storage (here located in the *MCT* component). Consequently, every different data unit should be associated with a separate external port. Thus the *MCT Rose-RT add-in* contains the collection of all external ports, which are used for communication with Matlab/Simulink via the *MCT*. The collection serves as a means for notification about the availability of data for each external port, i.e. an external port associated with some data is triggered by the *Remote control interface* to inform the port about data availability. The data itself is stored in the *MCT* component.

In order to allow the Matlab/Simulink model to store the data in the *MCT* component, knowledge about external ports and data types associated with these external ports should be available.

The *MCT Simulink add-in* is a separate Simulink block which should be added to an *Original Simulink model*.

The *MCT i*s a component, which provides the following three interfaces:

- The *Remote control interface* allows driving the Rose-RT execution in the step-by-step mode, i.e., it automates step-by-step execution.
- The D*ata interface* includes the functionality for data exchange between the models. For example, data calculated by Simulink is set in the *MCT* to be available for Rose-RT. After the *MCT* notifies Rose-RT about the data availability through the *Remote control interface,* Rose-RT can access the data in the *MCT* storage. The other direction of data transfer proceeds similarly. The *Data interface* also plays an important role in the time synchronization process. After executing a transition, the assumed execution time is sent to the *MCT Simulink add-in* using the *Data interface.* The role of the *MCT*

Simulink add-in is to pass the execution delay to the *Original Simulink model* and ensure that the data values are used after the right delay.

- The *Timing interface* stores the simulation time of Simulink and redirects timing requests from *Original Rose-RT model* to the Simulink timing.

5. Details of the Coupling

This section contains more implementation details of the realized coupling. The module architecture is described in Section 5.1. The component architecture, with an overview of all implemented components can be found in Section 5.2. Finally, Section 5.3 illustrates how a common notion of time has been established.

5.1 Module Architecture View

Figure 6 shows a more detailed module architectural view of the implementation, showing for instance in more detail how the timing of a UML model is obtained from the MCT. Below we describe the structure of the main parts, the Rose-RT layers, the MCT, and the Matlab/Simulink layers.

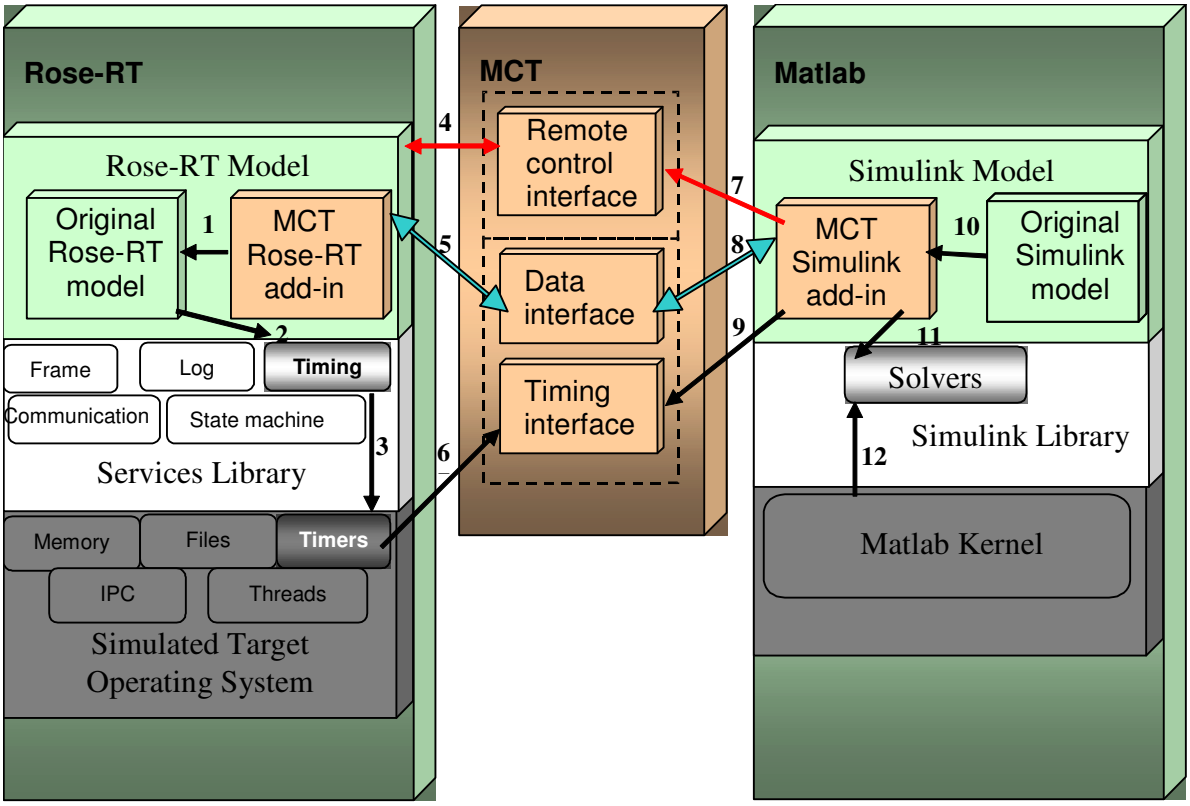

Figure 6 Module Architecture View

Rose-RT layers

The Rose-RT component consists of three layers: the Rose-RT Model layer, the Services Library layer, and the Simulated Target Operating System layer.

The Rose-RT Model layer consists of the *Original Rose-RT model* and the *MCT Rose-RT add-in,* which is responsible for the external communication (arrow 1 in Figure 6).

A Rose-RT model has access to classes of the *Services Library* layer. For example, whenever a model needs to use the timing service, as in the case of a timer creation, it uses the methods of classes that implement the *Timing Service* of the *Services Library* (see arrow 2).

Arrow 3 illustrates the dependency on target specific properties. For example, in case of a timing property request from the Rose-RT model, the *Timing Service* of the *Services Library* queries the *Timers* of the *Simulated Target Operating System* to obtain the requested property. This means that Rose-RT uses the timing properties of the operating system on which it is installed. In order to ensure the correct timing behavior of Rose-RT in the simulated environment, where it cooperates with Matlab/Simulink, the timing service used by Rose-RT has been adapted. This involves the creation of the *Simulated Target Operating System* and redefining the classes that implement the *Timing Service* of Rose-RT. The *Timers* of the *Simulated Target Operating System* should use the *Timing interface* of the MCT (see arrow 6) in order to access the simulation time of Simulink instead of the one originally used by Rose-RT.

MCT
As explained before, the MCT consists of three interfaces: a *Remote control interface,* a *Data interface*, and a *Timing interface*.

The *Remote control interface* allows starting, stopping and controlling the execution of the Rose-RT model in step-by-step mode (arrow 4). This functionality can be accessed by the *MCT Simulink add-in* (arrow 7). The *Data interface* serves as storage for the data that has to be exchanged between Rose-RT and Matlab/Simulink models, including the timing delays associated with the execution of transitions in Rose-RT (arrows 5 and 8).

The *Timing interface* keeps track of the simulation time. It represents an intermediate clock, which is updated with the value of the Simulink simulation time (arrow 9) and which is regularly sampled (before a step in Rose-RT is executed) by the *Timers* of the *Simulated Target Operating System* (arrow 6).

Matlab/Simulink layers
The Matlab/Simulink component has three layers: the Simulink Model layer, the Simulink Library layer, and the Matlab layer.

The Simulink Model layer contains the *Original Simulink model*, extended by the *MCT Simulink add-in* (see arrow 10). The *MCT Simulink add-in* is actually the driver of the simulation, and therefore the *Original Simulink model* dependents on it.

Timing of Simulink does not dependent on the platform on which the tool runs, but is defined by the *Solvers*. The *MCT Simulink add-in* takes the value of the current simulation time provided by one of the *Solvers* (see arrow 11) and passes it to the *Timing interface* (arrow 9).

The *MCT Simulink add-in* uses the functions of the *Remote control interface* to send events to the *MCT Rose-RT add-in*, and to drive the Rose-RT execution in the step-by-step mode (arrow 7). The command to perform a step in Rose-RT should always be preceded by an update of the Rose-RT time in order to keep the clocks of Rose-RT and Simulink synchronized.

It should be mentioned that after each step performed by Rose-RT, the *MCT Simulink add-in* gets the new data, including the assumed time duration of the executed transition(s). This data is obtained through the *Data interface*. To ensure that Simulink takes this execution delay into account, we have designed a block diagram in which the original Simulink model should be inserted. This block diagram will be discussed in more detail in Section 5.2.

5.2 Component Architecture View

This chapter describes in detail the design of the MCT components. Specifically, every package of the package diagram, presented in Figure 7, is addressed in a separate subsection. Observe that the MCT is

implemented by two separate packages, the *MCTtoRoseRT* and the *MCTtoMatlab* packages; they represent the interface between the *MCTatRoseRT* and *MCTatSimulink* packages.

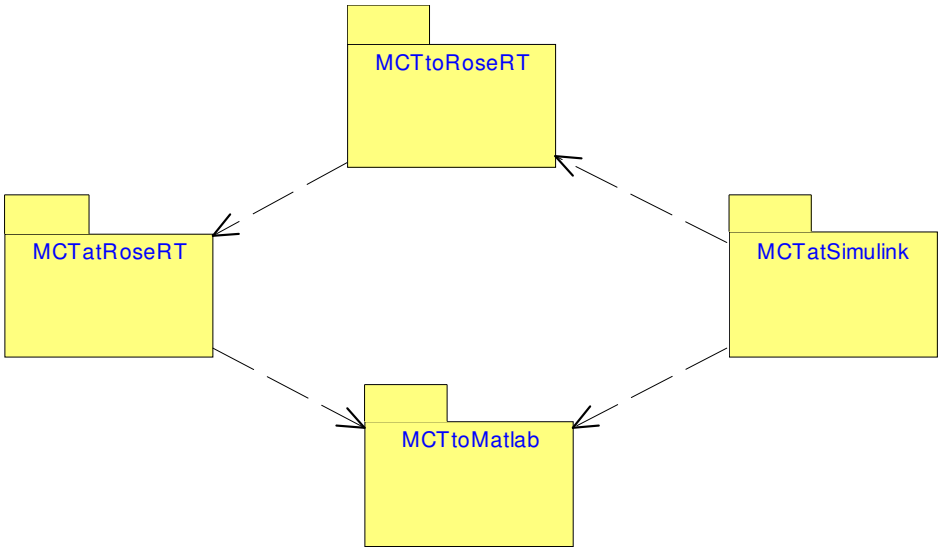

Figure 7 Package view

Package MCTatRoseRT

The content of the *MCTatRoseRT* package locates is combined with the original Rose-RT model. The class diagram of this package is depicted in Figure 8 where the *RoseRTmodel* represents the original model. It is extended with two classes, the *MCTClass* and the *PortClass*.

- The *PortClass* class, which represents a data structure with two fields: *portname*, the name of an external port, and *portinst*, a reference to the respective external port.
- The *MCTClass* class contains the collection of external ports, called *PortCollection*. The *raiseEvent()* function takes an external port as parameter and raises an event on it.

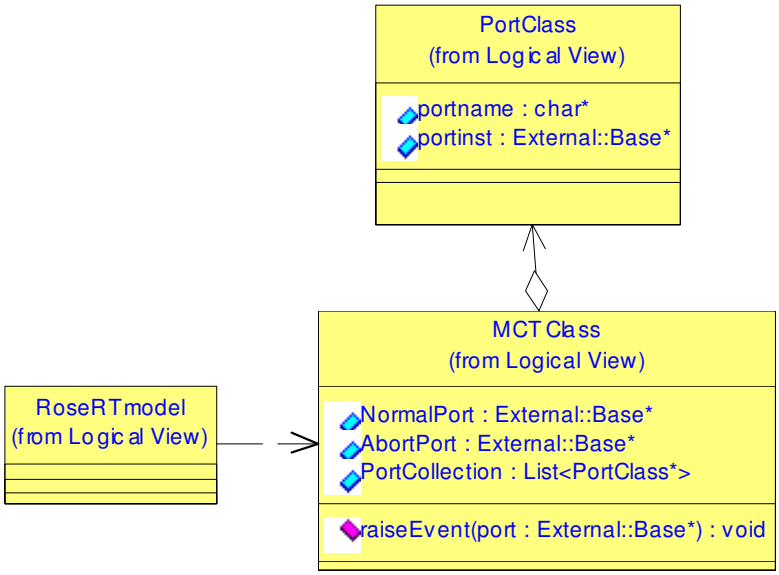

Figure 8 Package MCTatRoseRT - Class diagram

The user can fill in the collection of external ports in the MCTClass during the design of the software model. Suppose a user needs to create a capsule with an external port for an external communication. This requires opening a state diagram and declaring on the initial transition an instance of PortClass with a port name, which will be used as an identifier of the port from outside, and with a reference to the created external port. Next the user should add the instance of the *PortClass* to the *PortCollection*.

Package MCTtoRoseRT

Figure 9 shows the class diagram of the *MCTtoRoseRT* package. The *MCTtoRoseDynll* class consists of six functions, explained below, which provide the functionality for controlling the execution of the Rose-RT model. This functionality is applicable to the dynamic link library generated from the Rose-RT model. All variables and operations exported from the Rose-RT model need to be imported in *MCTtoRoseDynll*. Since an external port is not of a standard type, but specific only to the Rose-RT environment, it is not available in *MCTtoRoseDynll*. Therefore, for a successful import of the variables of the type of an external port, a class *PortClass* with the same structure as used in the *MCTatRoseRT* package has been introduced.

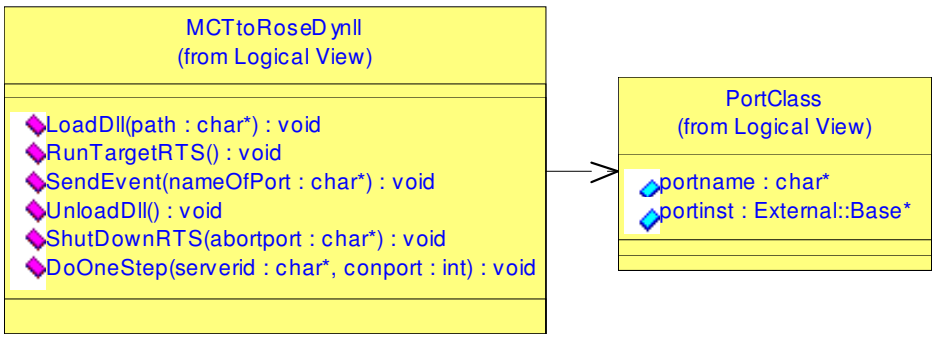

Figure 9 Package MCTtoRoseRT - Class Diagram

Class *MCTtoRoseDynll* contains the following functions:

- *LoadDll()* takes as parameter a full path to the location of the dynamic link library, generated from the software model and loads it. As a result, all exported variables and operations can be accessed within the *MCTtoRoseDynll* class.
- *RunTargetRTS()* creates an execution thread and starts a TargetRTS, making the software model ready for execution.
- *SendEvent()* takes as parameter the name of the external port, searches for the corresponding instance of the port in the port collection, and sends an event to it. Such an event notifies about the availability of data. As a result of receiving an event, a message is put into the message queue of the Rose-RT thread started by *RunTargetRTS()*. This message will be processed when a step has been performed.
- *UnloadDll()* cleans up the memory allocated by the library.
- *ShutDownRTS()* stops the TargetRTS.
- *DoOneStep()* forces the Rose-RT environment to perform a step. This may result in a state change if a transition is triggered. The trigger can be external (coming from outside of the model) or internal (coming from within the model).

Package MCTtoMatlab

The class structure of the *MCTtoMatlab* package is depicted in Figure 10. The *MCTtoMatlabDynll* class contains the functions for data manipulation and the transfer of the Simulink simulation time to Rose-RT. To avoid limitations on the amount of data types used, a collection of data is created. The attributes of the *Data* class are private and are accessed only through the functions which set and get the value of the attributes.

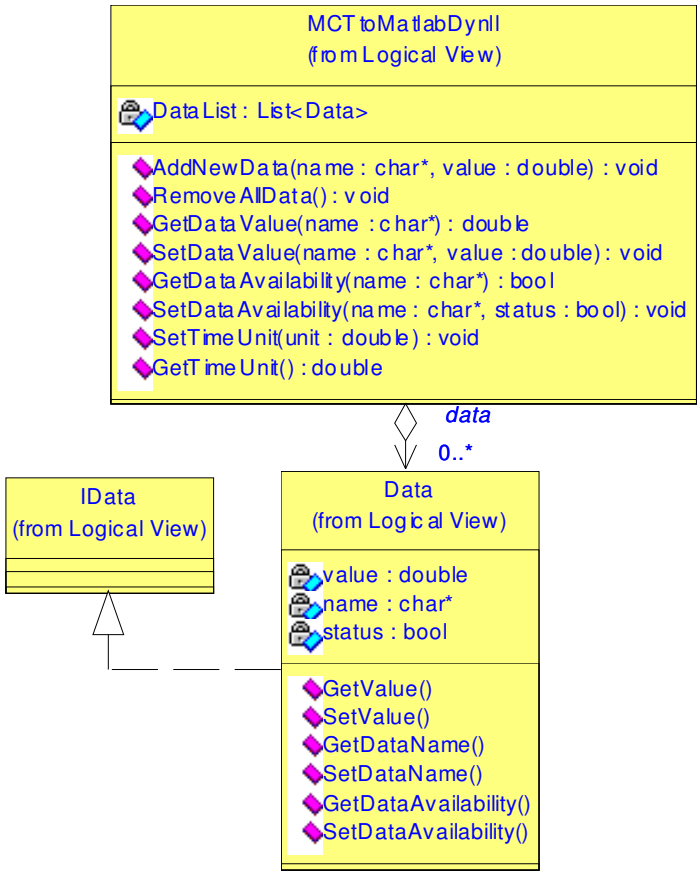

Figure 10 Package MCTtoMatlab - Class diagram

To transfer the duration of transition execution to Matlab, the user has to declare a variable that will contain the duration of the last executed transition. This variable should be added to the data collection with the help of the *AddNewData()* function.

Package MCTatSimulink

The *MCTatSimulink* package, as depicted in Figure 11, contains the *MCTSfunction*. This is implemented as an S-function, which defines a Simulink block and can be written in MATLAB, C, C++, Ada, or Fortran.

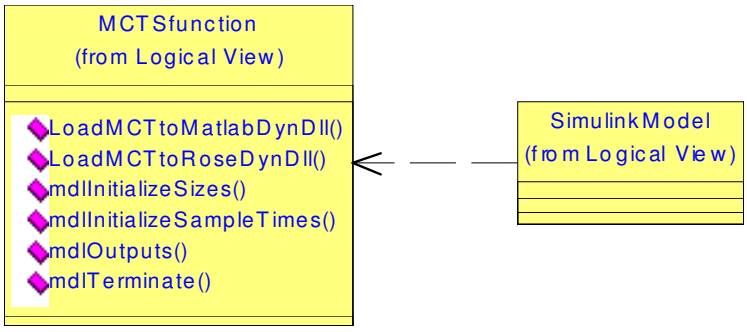

Figure 11 Package MCTatSimulink - Class diagram

The *LoadMCTtoMatlabDynll()* and *LoadMCTtoRoseDynll()* functions are called during the initialization of the *MCTSfunction*. Their purpose is to load the functions provided by the data and timing interfaces of the *MCTtoMatlab* package and of the remote control interface contained in the *MCTtoRoseRT* package.

Figure 12 shows the Simulink model prepared for coupling. It contains the block *Original Simulink model* and an additional set of other blocks which represent the *MCT Simulink add-in.*

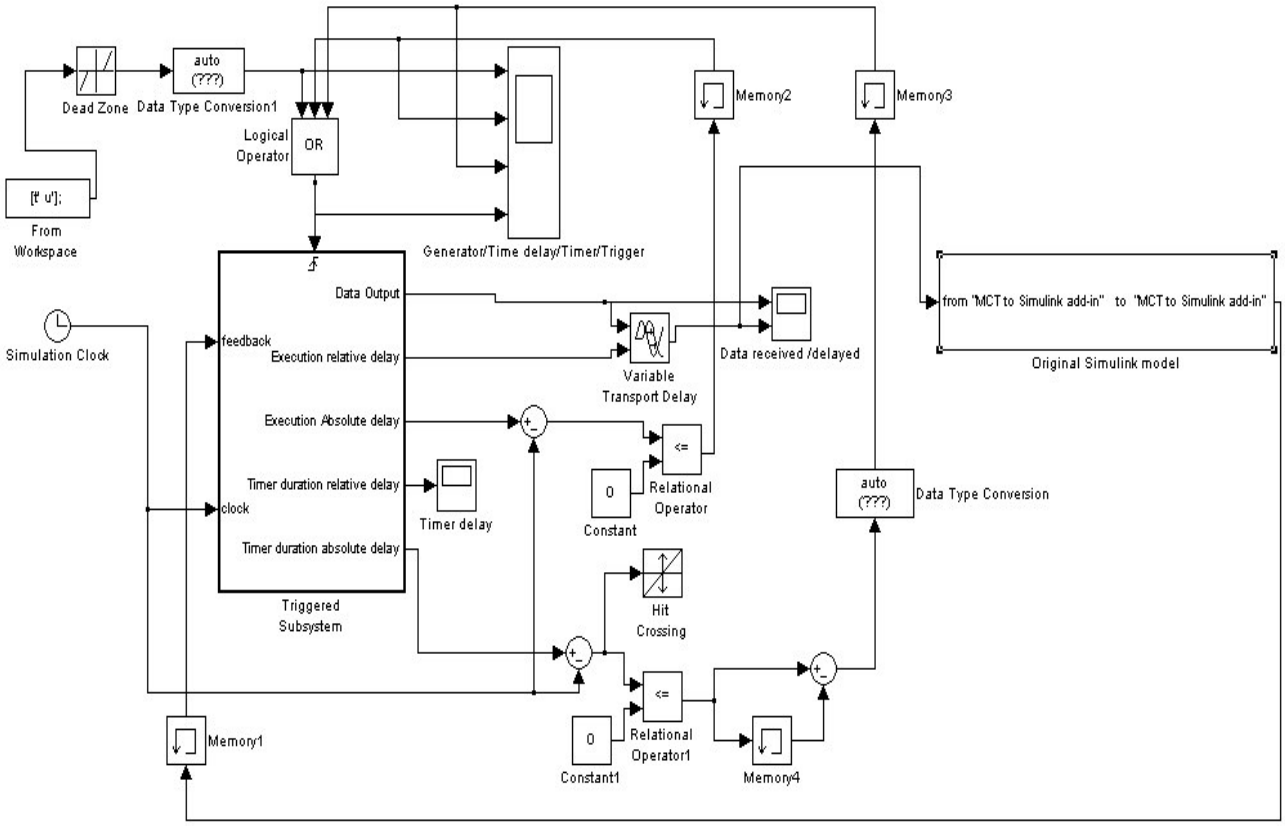

Figure 12 Simulink model with triggered subsystem

The *MCTSfunction* is incorporated into the *Triggered Subsystem* block (see Figure 13). The functionality of this block is enabled each time a trigger is detected. There are three types of triggers possible:

1. *Initial trigger*. It triggers the subsystem when the simulation starts. This trigger is represented by the blocks *From Workspace, Dead Zone*, and *Data Type Convertor.*
2. *Execution delay trigger*. This trigger is generated when the Simulink simulation time is advanced with the value of the execution delay associated with a transition executed in Rose-RT. It is represented by the blocks *Clock, Relational Operator*, and *Memory2*.
3. *Timer expiration trigger*. This trigger is generated when the Simulink simulation time is advanced with a value equal to the duration of the timer, as requested by a timer setting in Rose-RT. This trigger is represented by the blocks *Clock, Relational Operator, Hit Crossing, Memory3*, and *Memory4.*

If any of these triggers has been detected, the blocks contained in the Triggered Subsystem are executed in the order determined by the Simulink environment. Looking inside the *Triggered Subsystem*, as shown in Figure 13, one can notice an *MCTblock* which is a user-created block defined by the S-function *MCTSfunction.* This *MCTSfunction* has three outputs, which are passed to the five outputs of the *Triggered*

Subsystem block: *Data Output*, *Execution relative delay*, *Execution Absolute Delay*, *Timer duration relative delay*, and *Timer duration absolute delay*.

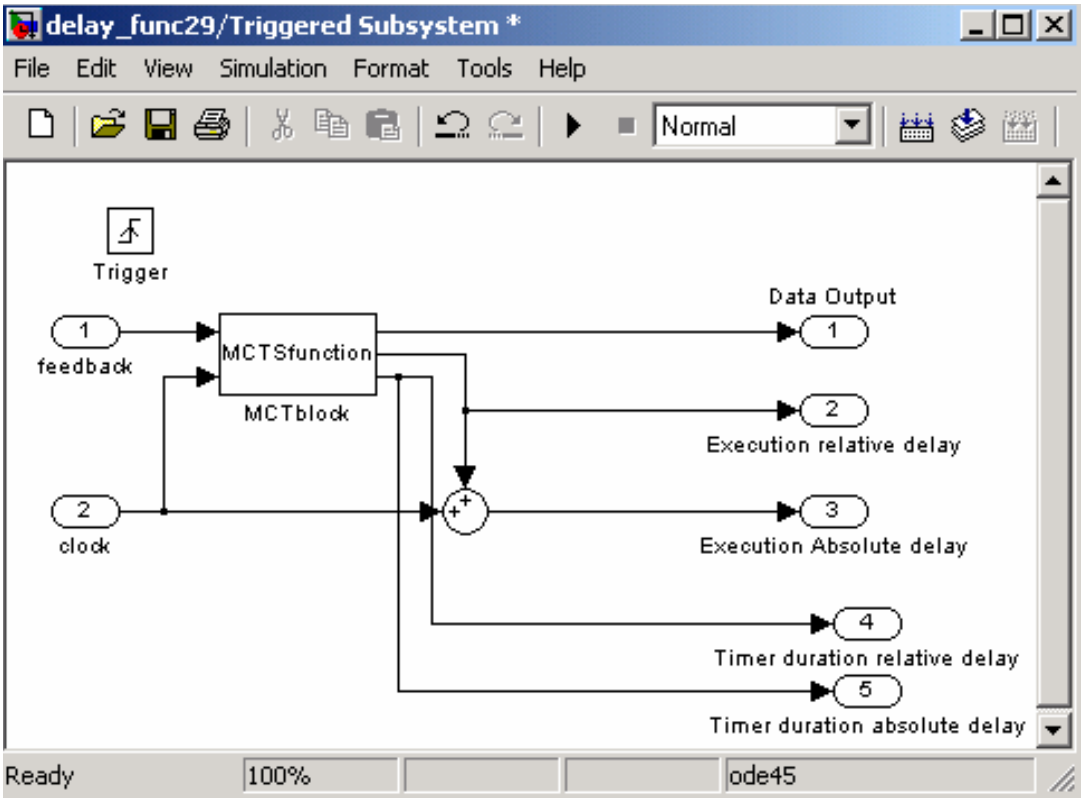

Figure 13 Triggered Subsystem

The E*xecution relative delay* is used for the simulation of delays by the *Variable Transport Delay* block (see Figure 12). This block ensures that data received from the *Data Output* of the *Triggered Subsystem* is provided only after the time specified by the *Execution relative delay*. The *Execution relative delay* is also used for the calculation of the *Execution Absolute delay*, which is used to generate *the Execution delay trigger* to the *Triggered Subsystem*.

Next we describe the behaviour of the S-function which implements the *MCTSfunction* defining the *MCTblock*. A standard S-function contains a number of functions that are called by the Simulink. For instance, Simulink calls *mdlInitializeSizes()* to inquire about the number of input and output ports, sizes of the ports, and any other objects (such as the number of states) needed by the S-function. During a simulation step, Simulink calls *mdlUpdate()* to update discrete states, *mdlDerivatives()* to calculate derivatives, and *mdlOutputs()* to calculate the outputs of a block. Finally, the mandatory *mdlTerminatePerform()* performs tasks at the end of the simulation.

In the following we focus on the *mdlOutputs()* function, which is called repeatedly. This function defines when and how the synchronization of the Rose-RT and Matlab/Simulink models is performed. The order in which commands are given within the *mdlOutputs()* function is crucial for the execution and should be strictly followed:

1. Set the clock in the *Timing interface* to the current simulation time. This is needed for synchronizing the Rose-RT clock with the Simulink clock.
2. Give a command to Rose-RT to perform one step. By doing this, Rose-RT will become active and responsive to external events.

3. Send an external event, if no timer was started in Rose-RT, which will trigger a transition in the Rose-RT model.
4. Read the data, execution delay and timer duration from the Data interface and pass it to the output of the *MCTblock.* If these were not set during the execution of the step in Rose-RT the previous values will be read.
5. Take the input of the *MCTblock,* provided by the *Original Simulink model* and put it in the *Data interface* to make it available for Rose-RT.

Finally, we propose the selection of a continuous variable-step solver for the simulation, to ensure an effective way of calculating data and determining critical points in the simulation.

5.3 Establishing a common notion of time

In this section, we illustrate how a common notion of time is established during the simultaneous simulation. The sequence diagram of Figure 14 shows the collaboration between various components during time synchronization. The figure consists of three parts: *Environment Initialization, Time Initialization*, and *Time synchronization*, and for each part it reflects the logical ordering of the function calls, i.e. the order in which the events should happen during the time synchronization process.

The *Environment Initialization* part reflects the steps needed for the initialization of the simulation environment. The *LoadMCTtoMatlabDynll()* and *LoadMCTtoRoseDynll()* functions are called to allow access to the functions of the MCT interfaces. The *LoadDll()* function is called to allow access to the variables and operations exported from the Rose-RT model.

The *Time Initialization* part reflects the steps needed to ensure that the Matlab/Simulink and the Rose-RT models, have the same notion of time at the start of the simulation. The *SetTimeUnit()* function is called with value 0 to start a simulation time counter that will be updated at each simulation time step. Next a call to the *RunTargetRTS()* function starts the execution of the Rose-RT model.
During the execution of a Rose-RT model, the current time is returned by the *getclock()* function of the *RTTimespec* class, which is part of the Rose *Services Library*. As mentioned in Section 5.1, the classes that implement the Timing Service of the *Services Library* have been redefined. In particular, function *getclock()* now uses the *GetTimeUnit()* function, which is provided by the *MCTtoRose* package and returns the stored simulation time.
Finally, function *AddNewData()* is called to add a variable (*Duration*) to the data collection; this is used to transfer the execution time of transitions in Rose-RT to Simulink.

The *Time synchronization* part reflects the steps needed to enforce a common simulation time. Before performing a step in Rose-RT the simulation time counter in *MCTtoMatlab* has to be set to the current simulation time, calling *SetTimeUnit(CurrentSimulationTime)*. When a step is performed in Rose-RT by a call of the *DoOneStep()* function, first the *GetTimeUnit()* function is called, using the redefined *getclock()* function. This will ensure that Rose-RT has the same notion of time as Matlab/Simulink.

At the end of the step, the execution time is transferred to the data interface by calling the *SetDataValue()* function (together with the produced data values). This duration is obtained by MCTatSimulink by calling the *GetDataValue()* function and next the appropriate delay is taken into account in Matlab/Simulink (represented by *advance simulation time with Duration*).

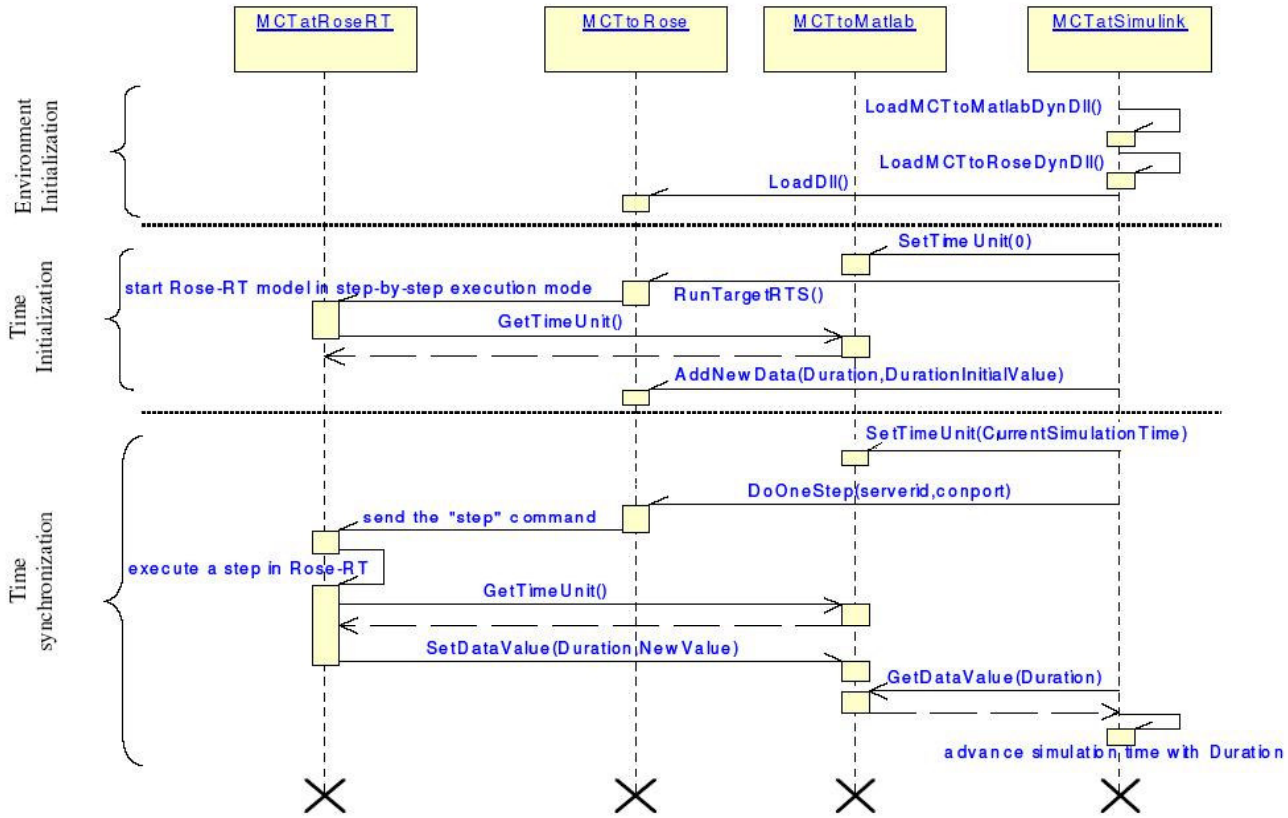

Figure 14 Time Synchronization Sequence Diagram

6. CONCLUDING REMARKS

The current version of the coupling tool that connects Simulink and Rose-RT is a first prototype that can be used to investigate the main principles and to experiment with examples. It is has been tested an a few small examples, but more experiments are needed to investigate the behavior for various types of solvers and models and to get more confidence in the correctness of the simulations. Moreover, we have to apply the coupling to large existing models from industry to investigate the feasibility and usefulness of such a simultaneous simulation and to investigate the performance for complex systems. Future work also includes the removal of a few simplifications that have been made to obtain a first prototype quickly. For instance, at the moment only one timer is allowed in the UML model and a preliminary version of a timer queue has not yet been tested.

Currently, the models are simulated on a single PC, but it might also be interesting to investigate a distributed implementation, which couples models on different PCs. Another possible topic of future work is the coupling with other UML-tools in the embedded systems domain, such as Telelogic Tau, Rhapsody of I-Logix, or Real-Time Studio of Artisan.

ACKNOWLEDGEMENTS

We would like to thank the members of the Boderc project for constructive discussions, useful hints for the realization of the coupling, support on the design of Matlab/Simulink models, and constructive comments on draft versions of the current paper.

REFERENCES

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

[2] E.M. Clarke, A Fehnker, Zhi Han, B. Krogh, J. Ouaknine, O. Stursberg, M. Theobald. *Abstraction and Counterexample-guided Refinement of Hybrid Systems.* International Journal of Foundations of Computer Science, Vol 14, Number 3, 2003.

[3] J. Dahmann, R. Fujimoto, and R. Weatherly. *The Department of Defense High Level Architecture* In Proceedings of the 1997 Winter Simulation Conference, pp.142-149, ACM, 1997.

[4] The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/

[5] R. Grosu, T. Stauner, M. Broy. *A Modular Visual Model for Hybrid Systems. In Proc. of the FTRTFT'98, 5th International School and Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems, Lyngby, 1998.*

[6] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. *HyTech: A Model Checker for Hybrid Systems.* Software Tools for Technology Transfer 1:110-122, 1997.

[7] The Ptolemy Project, http://ptolemy.eecs.berkeley.edu/

[8] Rose Technical Developer, http://www-136.ibm.com/developerworks/rational/products/rosetechnicaldeveloper/

[9] B. Selic, G., Gullekson, P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[10] Simulink of The Mathworks, http://www.mathworks.com/products/simulink/

[11] Simulink manuals, http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/

[12] T. Stauner, A. Pretschner, I. Péter, *Approaching a Discrete-Continuous UML: Tool Support and Formalization.* Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods -- Countering or Integrating the eXtremists, pp. 242-257, 2001.

[13] *TrueTime* source code and documentation, http://www.control.lth.se/~dan/truetime/