A Mechanism for Data Interchange Between Embedded Software Sub-systems Developed using Heterogenous Modeling Domains

Padma Iyenghar[1], Benjamin Samson[2], Michael Spieker[3], Arne Noyer[3], Juergen Wuebbelmann[2], Clemens Westerkamp[2] and Elke Pulvermueller[1]

[1]*Software Engineering Research Group, University of Osnabrueck, Osnabrueck, Germany*

[2]*Institute of Computer Engineering, UAS Osnabrueck, Osnabrueck, Germany*

[3]*Willert Software Tools GmbH, Hannoversche Str. 21, Bueckeburg, Germany*

Keywords: Embedded Software Sub-systems, Model-based Embedded Software Development, Heterogenous Modeling Domains, UML, Matlab/Simulink.

Abstract: In the domain of embedded systems, the complexities involved in embedded software development are being successfully addressed by the emerging field of model-based software development and testing. However, in embedded systems, the underlying embedded software is often expected to collaborate with various hardware, mechanical, electrical modules/technologies. To address this aspect of heterogeneity in embedded systems, practitioners of model-based embedded software engineering are required to use more than one modeling language. This is essential to address the multi-faceted design aspects/requirements of an embedded system. This paper elaborates on the existing data interchange and coupling mechanisms between embedded software sub-systems modeled using UML and Matlab/Simulink. While there are some existing coupling mechanisms for data exchange among heterogenous modeling domains, they are all not applicable to all real-time operating systems and/or limited to a few simulation studies. This paper addresses the aforementioned gaps and proposes a simple, generic methodology for data exchange between events (in UML domain) and signals (in Matlab/Simulink domain). The proposed approach is elaborated using a seesaw (real-word) embedded software system application scenario example. Initial prototype implementation of the proposed approach, experimental results and some future directions are outlined.

1 INTRODUCTION

Model Driven Development (MDD) introduced by the Object Management Group (OMG) (Object Management Group, 2014), is regarded as the next/ongoing paradigm shift in embedded software development. It is deemed as a key for fast, error-free and automated development of embedded software systems. On the other hand, heterogeneity seems to be the essence of all embedded systems: comprising of a subtle combination of hardware and software sub-systems for specifying data and control processing requirements. Further, the embedded software developed is often expected to execute a feedback mechanism and/or exchange updated data and respond to various external sensors/stimuli (e.g. mechanical, electronic modules) (Vanderperren et al., 2012). This implies that, embedded software developers adopting a MDD approach are required to employ more than one modeling language to address the multi-faceted design aspects of an embedded software system. Thus, an embedded software system may comprise of several software sub-systems, developed using heterogenous modeling domains (Fig. 1).

Let us consider a simple heterogenous embedded system comprising of two main sub-systems. One sub-system is used for modeling the application level aspects and the other sub-system is used for dealing with the low-level/hardware-related continuous functions. At the application level, the system architecture could be modeled, for instance, using UML class diagrams. Similarly, the discrete/event-based functionality, at the application level, may be modeled using state machines. The application level sub-system may be considered to be less time critical. Whereas, the sub-system used to model the low-level/hardware-

Iyenghar P., Samson B., Spieker M., Noyer A., Wuebbelmann J., Westerkamp C. and Pulvermueller E..
A Mechanism for Data Interchange Between Embedded Software Sub-systems Developed using Heterogenous Modeling Domains.
DOI: 10.5220/0005335604190424
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 419-424
ISBN: 978-989-758-083-3

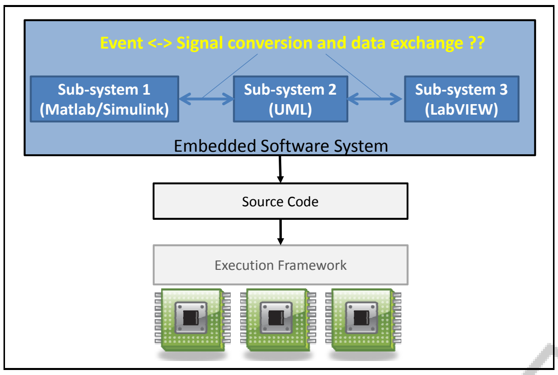


Figure 1: Embedded Software Development Using Heterogeneous Modelling Domains.

related aspects, handles the continuous signals and contains more time critical tasks. This could be modeled, for instance, using Matlab/Simulink (Matlab and Simulink, 2014) or LabVIEW (LabVIEW System Design Software, 2014)) (Fig. 1).

Data exchange among these sub-systems is a critical and challenging task. To bring the requirements of both the levels together and enable bi-directional data exchange, a coupling concept must be developed. A key aspect in bi-directional data exchange, is the conversion (of data) from events to signals and vice versa. In the existing literature and tool support, some coupling concepts are discussed and supported (Nicolescu et al., 2012), (Hooman et al.,). However, such coupling methods are not applicable to all real-time operating systems and/or limited to a few simulation studies.

This paper addresses the aforementioned gaps and demonstrates a novel methodology to convert events to signals and vice versa. The main contribution of this paper is the proposal of a simple and a generic mechanism for bi-directional data exchange among sub-systems modeled using heterogenous modeling domains, e.g. UML and Matlab/Simulink.

The remaining of this paper is organized as follows. Related work pertaining to data exchange/coupling between UML and Matlab/Simulink domains is discussed in section 2. A seesaw model example, developed using heterogenous modeling domains is introduced in section 3. The proposed approach and initial results are discussed in section 4. Section 5 presents a conclusion.

2 BACKGROUND AND RELATED WORK

Related work pertaining to alternatives for data exchange and coupling between Matlab/Simulink and UML domains, used for modeling heterogenous embedded software systems, is discussed in this section.

2.1 Coupling UML and Matlab/Simulink

Coupling the execution of UML and Matlab/Simulink models can be achieved by (a) co-simulation, (b) model-level coupling and (c) source-code level coupling.

In the case of co-simulation, both the UML and Simulink tools are linked by a coupling tool (Vanderperren et al., 2012). However, in this approach, special attention to a consistent notion of time is crucial to guarantee proper synchronization between the UML tool and Matlab/Simulink (Vanderperren and Dehaene,) (Nicolescu et al., 2012) (Hooman et al.,).

Model-level coupling of heterogenous embedded systems discussed in (Reichmann et al., 2004) uses model-to-model transformation and bi-directional transformation rules. On the other hand, such model-to-model transformation of large system architecture during the development process is susceptible to errors. Further, scalability of such model-based coupling of software components and time-discrete/continuous parts for a real-life industrial case study is missing. Scalability issues of the approach proposed in (Reichmann et al., 2003) can be attributed to the lack of a commercial version of the tool *GeneralStore* outlined in (Reichmann et al., 2004).

The third alternative for coupling the execution of UML and Matlab/Simulink models is integration based on the underlying source code or executable language. This approach can be considered more advantageous, for small and medium projects. This methodology paves the way for integration of time-discrete and time-continuous sub-systems at the source code level, without having to consider coupling the entire sub-system at the source-code or model-level.

A combination of the model-level and source-code level coupling between the Matlab/Simulink and UML domains is available in a MDD tool Rhapsody (IBM Rational Rhapsody Developer, Ver 8.4, 2014). Integration of Matlab/Simulink with Rhapsody can be performed by using a feature called ”SimulinkBlocks” supported by Rhapsody. In this method, blocks (e.g. objects created in a given Rhapsody project) are stereotyped with ”SimulinkBlock” stereotype. Then the required Matlab/Simulink files are imported to this object. Once the import is successful the values to be exchanged between Matlab/Simulink and Rhapsody are assigned as *flowports* (a feature in Rhapsody) in these objects. The coupling mechanism using the Systems Modeling Language (SysML) follows a similar path (in MDD tool (IBM Rational Rhapsody Developer, Ver 8.4, 2014)). Both these methods depend on the usage of *flowports* as interaction points and interfaces between blocks (Systems Modeling Language (SysML), 2014); a feature which cannot be supported by all real-time operating systems.

Thus, in the existing literature/tool support, a generic methodology for data interchange between events and signals (time discrete and time continuous domains) is not yet available. This paper elaborates on this aspect and proposes a simple, generic mechanism for data exchange between UML and Matlab/Simulink domains.

3 A SEESAW MODEL EXAMPLE

In this paper, a seesaw application (Fig. 2) is used as a real-life embedded system example for the evaluation of the proposed approach.

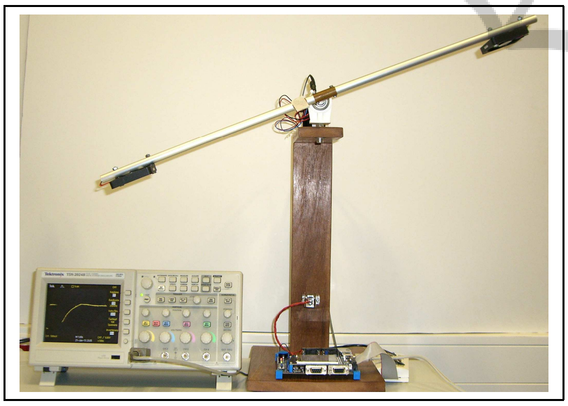


Figure 2: Seesaw model.

3.1 Construction and Design

A seesaw is equipped with two ventilators, one at each end. One ventilator produces a momentum in clockwise direction, the other in anti-clockwise direction, to accelerate the seesaw. The speed of the ventilators is controlled by Pulse Width Modulation (PWM). The corresponding/current angle of the seesaw is measured with a potentiometer and converted with an Analog-To-Digital Converter (ADC). The seesaw application moves the seesaw to different angles, one after another.

3.2 Implementation

In this example, the discrete modeling domain (modeled using UML in MDD tool Rhapsody (IBM Rational Rhapsody Developer, Ver 8.4, 2014)), is responsible for setting the desired angle of the seesaw. This sub-system output (angle of the seesaw) is transferred to a global variable and controlled by a closed loop control. The closed loop control sub-system is implemented in the Matlab/Simulink modeling domain (Fig. 3). The PID control unit is generated with the MATLAB/Simulink Embedded Coder. It handles the parameters received from the discrete modeling domain. Fig. 4 shows the implementation of the PID controller in MATLAB/Simulink. It also describes the mapping between ADC values and degrees for the ventilator. The output values are converted to PWM values.

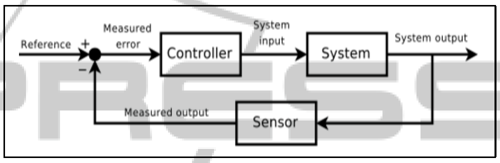
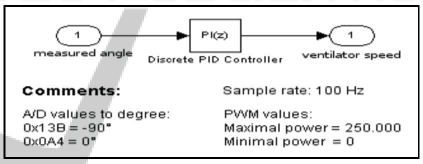


Figure 3: System control loop.



A/D values to degree:
0x13B = -90°
0x0A4 = 0°

PWM values:
Maximal power = 250.000
Minimal power = 0

Figure 4: PID loop controller.

The embedded software thus developed, runs on a Keil LPC1700 Board (Embedded development tools , 2014), which contains an ARM Cortex M3 CPU from NXP. The compiler is the Keil/ARM MDK using μVision. The RTOS used is OO-RTX single threaded UML based RTOS from (Willert Software Tools, 2014). Table 1 shows a simplified set of signals, events and examples of boundary conditions requiring event to signal conversion and vice versa. Further implementation aspects such as, double buffering for read/write data between the two domains, cyclic functions invoking control loop and measurement functions are not elaborated here.

4 PROPOSED APPROACH

In this section, a generic notation for describing a heterogenous embedded system is introduced. A set of procedures based on the generic notations are described, proposing a simple but effective methodology for data exchange between UML (discrete events) and Matlab/Simulink (continuous signals) domains. Examples of event to signal conversion and vice versa

Table 1: Events, Signals and boundary conditions in seesaw example.

Variable	Type	Description of the example
ADCValue	Signal	Seesaw angle, Output from UML domain
PWMValue	Signal	Ventilator speed, Output from M/S domain
evRelease	event	Used in event to signal conversion*
evReset	event	Used in signal to event conversion$

* Boundary condition for **event to signal conversion**: on button press, generate (event) *evRelease* and set (signal) *ADCValue*.

$ Boundary condition for **signal to event conversion**: If *PWMValue* (signal) is greater than threshold, generate (event) *evReset*, to reset angle of seesaw.

are explained with the help of the seesaw model.

4.1 Generic Representations

Let us consider a heterogenous embedded software system $HE_{ss} = \{SS_1, SS_2, ..., SS_n\}$, where SS_1, SS_2, ..., SS_n are sub-systems modeled using various modeling domains such as UML, Matlab/Simulink and LabVIEW. In the seesaw example, $HE_{ss} = \{SS_1^{UML}, SS_2^{M/S}\}$, where SS_1^{UML} is the sub-system modeled using UML (discrete domain, Rhapsody tool) and $SS_2^{M/S}$ is the sub-system modeled using Matlab/Simulink (continuous domain, e.g: PID controller). In the context of this paper, let us consider only the events and signals in the UML and the Matlab/Simulink domains respectively. The events and signals can be represented as $SS_n^{UML} = \{e_n^1, e_n^2, e_n^3, ..., e_n^n,\}$ and $SS_n^{M/S} = \{s_n^1, s_n^2, s_n^3, ..., s_n^n\}$. Here, e_n^1, e_n^2, e_n^3 ... e_n^n are representation of the events in the sub-system modeled using UML (SS_n^{UML}). Similarly, $\{s_n^1,\ s_n^2,\ s_n^3,\ ...,\ s_n^n\}$ is a representation of the signals in the sub-system modeled using Matlab/Simulink ($SS_n^{M/S}$).

In UML, the *tag* value of an event can be used to specify additional information to the event. In our approach, consider that each event is associated with two *tag* values, one each for event to signal conversion and vice versa. Then, the tag values corresponding to e_n^1, e_n^2, ... e_n^n for *event to signal conversion* can be represented as $Tag\_e_n^1$**_EvtoSi**, $Tag\_e_n^2\_EvtoSi$, ..., $Tag\_e_n^n\_EvtoSi$. Similarly, the tag values corresponding to e_n^1, e_n^2, ... e_n^n for *signal to event conversion* can be represented as $Tag\_e_n^1$**_SitoEv**, $Tag\_e_n^2\_SitoEv$, ..., $Tag\_e_n^n\_SitoEv$.

4.2 Event to Signal Conversion

For event to signal conversion, let us consider a simple scenario: on an external button press (available in Keil LPC1700 board (Embedded development tools , 2014)), the seesaw is expected to go to a new/next position. The idea here is that on a button press, an event (e.g: *evRelease*) is generated, which in turn is expected to be converted/mapped to a signal. The signal value is the angle of the seesaw (e.g: signal name =*ADCValue*), managed by the PID controller (Fig 3, 4) in the Matlab/Simulink domain. Thus a mechanism for communication between the discrete and continuous domains is necessary for data exchange. Mapping the aforementioned examples to generic notations, the examples can be represented as $SS_1^{UML} = \{e_1^1 = evRelease\}$ and $SS_1^{M/S} = \{s_1^1 = ADCValue\}$ (Table 1).

A procedure for event to signal conversion is shown in algorithm 1. The inputs for this algorithm are the events $e_n^1, e_n^2, ..., e_n^n$ (UML domain) and their respective event to signal *tag*, $Tag\_e_n^1\_EvtoSi$, $Tag\_e_n^2\_EvtoSi$, ..., $Tag\_e_n^n\_EvtoSi$. The output of this algorithm is setter methods for signals w.r.t the corresponding events.

Algorithm 1: Event to Signal Conversion.

```
Input: $SS_n^{UML} = \{e_n^1, e_n^2, ..., e_n^n\}$ and $Tag\_e_n^1\_EvtoSi$,
    $Tag\_e_n^2\_EvtoSi$, ..., $Tag\_e_n^n\_EvtoSi$ ;events and
    tags
Output: Setter methods for event to signal conversion
1: if $SS_n^{UML} \neq 0$ then
2:    create $W = setterForSignals$ ;object/wrapper class
3:    create $St_W$ in $W$ ; new statechart with default transition
4:    create new events with name corresponding to tag value, as $ev\_Set|Tag\_e_n^1\_EvtoSi|$;
5:    create setter methods in newly created events, $ev\_Set|Tag\_e_n^1\_EvtoSi| = |Tag\_e_n^1\_EvtoSi|$
6: else
7:    exit
8: end if
```

The setter methods are created (as action values) corresponding to the events and tags in a newly created state chart (in a wrapper class/object), by the algorithm 1. The wrapper class and the statechart created inside the wrapper class are represented as $W = setterForSignals$ and St_W respectively. Please note that, in this paper it is assumed that in the statechart, an action is executed upon receiving a trigger (i.e., an event).

Let us consider our example event *evRelease* (e_1^1) with corresponding tag value as $Tag\_e_1^1\_EvtoSi = \{ADCValue = 20\}$. Given this input, the algorithm 1 creates an object/wrapper class (step 2) and a new state chart with default state and transition (step 3). The wrapper class and the state chart created inside the wrapper class are represented as $W = setterForSignals$ and St_W respectively.

In the next step (step 4), new events are added to the statechart created in the previous step. The events are created based on the tag values of the events given as input. For example, for the event $e_1^1 = evRelease\}$ with tag $Tag\_e_1^1\_EvtoSi = \{ADCValue = 20\}$, the newly created event is *evSetADCValue*.

In step 5, the algorithm creates setter method in the *action* field of the newly created event *ev_SetADCValue*. The action field value for *ev_SetADCValue* is $ADCValue = 20$. Thus algorithm 1 creates setter methods for the events in UML domain, for setting the corresponding signal values in Matlab/Simulink domain. It is up to the end-user to make use of these setter methods in the UML domain to invoke exchange of data values in the Matlab domain. For example, the event *ev_SetPWMValue* can be invoked in the action field of *evRelease* to set the signal value on button press on the target board.

4.3 Signal to Event Conversion

In the proposed approach, the required boundary condition for signal to event conversion is specified in the tag value ($Tag\_e_n^n\_SitoEv$) of an event (e_n^n) in the UML domain. A procedure for signal to event conversion is shown in algorithm 2. The inputs for this algorithm are the events $e_n^1, e_n^2, ..., e_n^n$ (UML domain) and their respective signal to event tag, $Tag\_e_n^1\_SitoEv$, $Tag\_e_n^2\_SitoEv$, ..., $Tag\_e_n^n\_SitoEv$. The output of this algorithm is the respective call to fire events, which are inserted inside of the step functions. This achieves signal to event conversion, in a simple but effective way.

As seen in algorithm 2, in step 1, the tag values of the signal to event tag in each event in the given UML sub-system is parsed. In our example, this can be represented as $e_1^2 = evReset\}$ (event) with $Tag\_e_1^2\_SitoEv = \{\text{FIRE\_IF\_RISING\_}systemInput\text{\_20}\}$ (tag value). The meaning of this tag is the following: fire an event (*evReset*) when the threshold value of the variable *systemInput* in UML domain (represents the value of the signal PWMValue) increases above the value 20.

Algorithm 2: Signal to Event Conversion.

```
Input: $SS_n^{UML} = \{e_n^1, e_n^2, ..., e_n^n\}$ and $Tag\_e_n^1\_SitoEv$,
    $Tag\_e_n^2\_SitoEv$, ..., $Tag\_e_n^n\_SitoEv$ ;events and
    tags
Output: Function calls to fire respective events
    (e.g. FIRE_IF_RISING(destination, event, signal,
    threshold))
1: if $SS_n^{UML} \neq 0$ then
2:     parse the value of signal to event tag,
       $Tag\_e_n^n\_SitoEv$
3:     create function calls based on tokens parsed
4:     Insert function call (inside step function) in
       UML domain, to fire respective events;
5: else
6:     exit
7: end if
```

Note that, as a first step in the prototype implementation, only two types of macros such as *FIRE_IF_RISING(destination, event, signal, threshold)* and *FIRE_IF_FALLING(destination, event, signal, threshold)* are implemented. These are used to support invocation of firing of events when the threshold of signals rise or fall below a specified threshold.

In step 2, function calls based on the tokens parsed in the previous step is created. For our example, the function call created is *FIRE_IF_RISING(destination (self), evReset, PWMValue, 20)*. In the final step, the function call thus created as a result of parsing the tag value is inserted in the the appropriate step function to facilitate signal to event conversion.

5 CONCLUSION AND FUTURE WORK

This paper describes a simple, but effective methodology for event to signal conversion and vice versa, allowing the coupling between heterogenous embedded software sub-systems. A generic notation for describing a heterogenous embedded software system is introduced. A set of procedures based on the generic notation are described, proposing a simple but effective mechanism for data exchange between UML (discrete events) and Matlab/Simulink (continuous signals) domains. A prototype of the proposed generic methodology and algorithms is implemented in the programming language Java with the aid of APIs (Rational Rhapsody API Reference Manual, 2014) in the MDD tool used (IBM Rational Rhapsody Developer, Ver 8.4, 2014). The prototype is evaluated in a seesaw model real-world embedded software system application scenario, modeled using heterogenous modeling domains such as UML and Matlab/Simulink.

Future directions include: supporting more than one controller for Matlab/Simulink domain and exploring possibilities to specify boundary conditions in UML (other than *tags*).

ACKNOWLEDGEMENT

This project is supported by a grant from the federal ministry of Economics and Technology (BMWi), Germany. This work is carried out in close cooperation with the industrial partner Willert Software Tools GmbH.

REFERENCES

Embedded development tools (2014). http://www.keil.com/.

Hooman, J., Mulyar, N., and Posta, L. Coupling simulink and UML models. In *Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMATS 2004*, pages 304 – 311.

IBM Rational Rhapsody Developer, Ver 8.4 (2014). http://www.ibm.com.

LabVIEW System Design Software (2014). http://www.ni.com/labview/.

Matlab and Simulink (2014). http://www.mathworks.com/.

Nicolescu, G., OĆonnor, I., and Piguet, C. E. (2012). *Design Technology for Heterogeneous Embedded Systems*. Springer.

Object Management Group (2014). http://www.omg.org.

Rational Rhapsody API Reference Manual (2014). http://www.ibm.com/.

Reichmann, C., Kuehl, M., Graf, P., and Muller-Glaser, K. (2004). GeneralStore - a CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004. Proceedings.*

Reichmann, C., Kuehl, M., and Maeller-Glaser, K. (2003). An overall system design approach doing object-oriented modeling to code-generation for embedded electronic systems. In Pezzae, M., editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin / Heidelberg.

Systems Modeling Language (SysML) (2014). http://www.sysml.org.

Vanderperren, Y. and Dehaene, W. From UML/SysML to Matlab/Simulink: Current state and future perspectives. In *Design, Automation and Test in Europe, DATE 2006, proceedings*.

Vanderperren, Y., Mueller, W., He, D., Mischkalla, F., and Dehaene, W. (2012). Extending UML for electronic systems design: A code generation perspective. In Nicolescu, G., O'Connor, I., and Piguet, C., editors, *Design Technology for Heterogeneous Embedded Systems*. Springer.

Willert Software Tools (2014). http://www.willert.de/.