GRAPHICAL FRAMEWORK FOR SYSTEM LEVEL DESIGN SPACE EXPLORATION

Klemen Perko[1], Andrej Trost[2]

[1]Sipronika d.o.o., Ljubljana, Slovenia
[2]Faculty of Electrical Engineering, Laboratory for Integrated Circuits Design, University of Ljubljana, Ljubljana, Slovenia

Key words: abstraction, design-space exploration, graphical modeling, high-level design, system-level simulation.

Abstract: As technology advances, options for realization of heterogeneous systems increase. Designers use a variety of hardware (HW) and software (SW) co-design methodologies in order to meet application constraints as fast as possible. The paper presents a graphical modeling framework used for high-level modeling and design-space exploration of heterogeneous systems. The framework provides designer graphical elements for using modeling concepts from system modeling libraries. Graphical modeling relieves the designer of the manual-typing source code and thus hides many details of system-level design languages that normally need to be taken care of. The graphical framework also provides different constraint checks during modeling and automatically generates an executable model for evaluation of a heterogeneous system. Our case study exemplifies the use of the framework and shows what information is obtained from an executable model built on a high-level of abstraction. Evaluation of results serves as a basis for further design decisions. Graphical modeling enables rapid changes in the model and thus speeds-up design-space exploration.

Grafično okolje za raziskovanje načrtovalskega prostora na nivoju sistemov

Kjučne besede: abstrakcija, modeliranje, raziskovanje načrtovalskega prostora, visokonivojsko načrtovanje.

Izvleček: S tehnološkim napredkom se povečuje nabor možnih realizacij heterogenih sistemov. Načrtovalci za čimprejšnje izpolnjevanje načrtovalskih zahtev uporabljajo širok spekter metodologij za sočasno načrtovanje strojne in programske opreme. Članek predstavlja grafično modelirno okolje za modeliranje in raziskovanje načrtovalskega prostora heterogenih sistemov. To okolje omogoča načrtovalcu uporabo grafičnih elementov pri modeliranju konceptov iz knjižnic za modeliranje na sistemskem nivoju. Grafično okolje načrtovalca razbremenjuje ročnega pisanja programske kode, tako da mu ni več potrebno poznati točne sintakse ukazov programskih jezikov za modeliranje na sistemskem nivoju. Okolje med izdelavo modela preverja njegovo skladnost z različnimi omejitvami. Po končanem modeliranju heterogenega sistema, za njegovo ovrednotenje okolje avtomatsko ustvari izvršljiv model. Uporaba okolja je prikazana na praktičnem primeru. Prikazano je, katere informacije dobimo iz izvršljivega modela zgrajenega na visokem nivoju abstrakcije. Ovrednoteni rezultati predstavljajo podlago nadaljnjim načrtovalskim odločitvam. Grafično modelirno okolje omogoča hitre spremembe modela in tako pospeši raziskovanje načrtovalskega prostora.

1 Introduction

Advances in technology provide various options for realization of embedded systems. Designers are encouraged to use a variety of HW and SW implementation technologies in order to meet application constraints and provide quick time-to-market solutions. The increasing complexity of modern embedded systems requires new design methodologies and system-level design tools /1/.

Many research studies are concentrated on the issues of HW/SW co-design, co-simulation and various optimization techniques. The research activity is slowly drifting away from modeling heterogeneous aspects of the system towards system description on a higher abstraction level /2/.

This paper will present a design framework for system-level design space exploration. The presented framework is used for a quick evaluation of design decisions in the first stages of the design process. The evaluation is based on the results obtained from a high-level model of the system composed in a graphical framework.

2 Design space exploration

HW and SW components of digital systems are designed by using specialized languages. The HW description language VHDL /3/ or Verilog is used for design and implementation of HW components and the C or C++ is used for SW description. These languages are mature and provide automatic implementation and various optimization possibilities.

On the system level, we need tools and languages for modeling systems composed of HW and SW components. The result of research in this area is several system level design languages (SLDL) and HW/SW co-design methodologies /4/.

A typical design flow starts with a high-level system model containing architecture description, functionality description and mapping information /5/. During design-space exploration, the model is repeatedly evaluated and changed until the application constraints are met. If the design methodology supports different levels of abstraction, we have

to repeat design-space exploration on each level. In each level we add new information thus lowering the level of abstraction. Finally, a description of HW and SW components prepared for automatic implementation tools is obtained.

2.1 SystemC Design Flow

The SystemC is a system-level description language based on the C++ language. The programming language C++ can be used also as an extensible object-oriented modeling language. The SystemC extends the capabilities of the C++ by enabling modeling of hardware descriptions /6/, /7/.

The SystemC language is implemented as a C++ class library. It adds important concepts to the C++ such as concurrent processes execution, modeling timed events and hardware data types. The SystemC enables designer to describe the whole system model in one language, verify it by using the same language, and further refine it all the way to the implementation level (typically the register transfer level). A system can be modeled at the behavioral or architectural level and then iteratively refined to the register transfer level.

Building a detailed model of an embedded system in the SystemC can be a very time-consuming task. In order to speed-up the design space exploration process, we need to identify important modeling concepts for the model evaluation at the current abstraction level. When a satisfactory model is obtained, more details can be described (for example timing and communication) and the design-space exploration is repeated on a lower level of abstraction /5/. Model refinement continues until all the details necessary for implementation are obtained.

In this paper we will focus on design exploration on the highest abstraction level. In the first stage of the design process we can identify some concepts repeatedly needed by designers for any new model. A model of an embedded system is composed of HW (architecture) and SW (functionality) units. SW units are running on the model architecture using its resources. SW can be further modeled as a composition of some tasks. A high-level architectural model contains execution (processing), communication and data storage units. To relieve the designer of the burden of repeatedly implementing models of these basic concepts in the SystemC, system modeling libraries supporting them were developed in our Laboratory /8/, /9/. They provide wrappers for modeling functionality and architecture on a high-level of abstraction.

System functionality is intuitively described as a network of tasks. The tasks are modeled in terms of architectural resource usage and no actual algorithm is specified. Each task is assigned an execution unit responsible for characterizing the cost of executing services (e.g. time, energy and size by means of logical blocks or transistors count). Execution and communication units are parts of the architecture description.

A library with functionality wrappers provides mechanisms for modeling parallel task execution. Event modeling is used for triggering task execution. From the functionality point of view, the task execution is limited only by their data dependency. The maximum level of parallel operations that a specific algorithm permits is first examined and later decreased by applying restrictions of execution units. When more than one task specifies the same execution unit, it is up to the execution-unit scheduling policy to determine the outcome of such request.

An architecture-wrappers library provides support for high-level modeling of architectural resources. Using these wrappers, designers can instantiate and connect any number of hardware units in their model and build an architectural model. Concepts of execution and communication units present HW resources and give the designer only information about architecture resource utilization in interaction with algorithm functionality.

One of the integral parts of our libraries is also a built-in support for logging relevant information about the system during execution of simulation. The system modeling libraries enable a component-based construction of the system model at a higher abstraction level. The concept of components promotes reuse of the already developed models which can leverage design productivity.

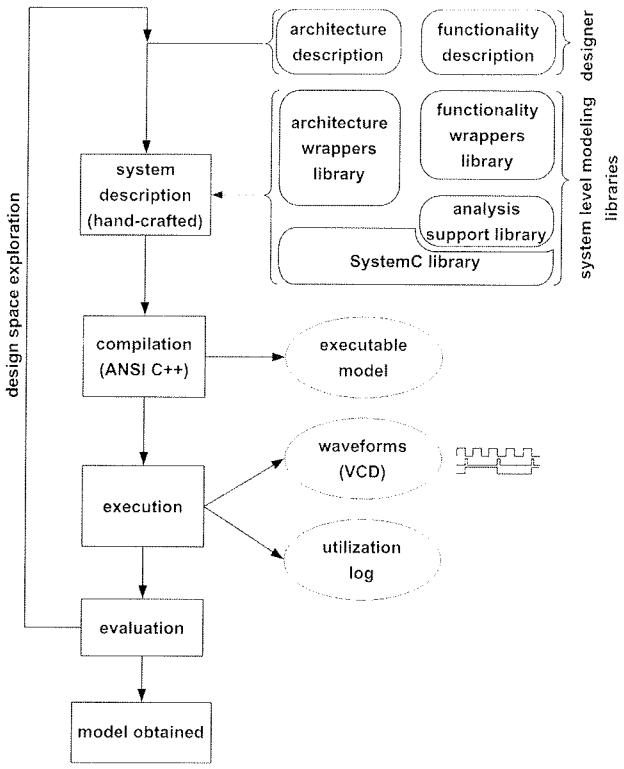

Fig. 1. Design flow using a system modeling library

The design-space exploration flow starts by composing a system model containing descriptions of architecture and functionality by using prepared wrappers from the system modeling library (see Figure 1). The model composition

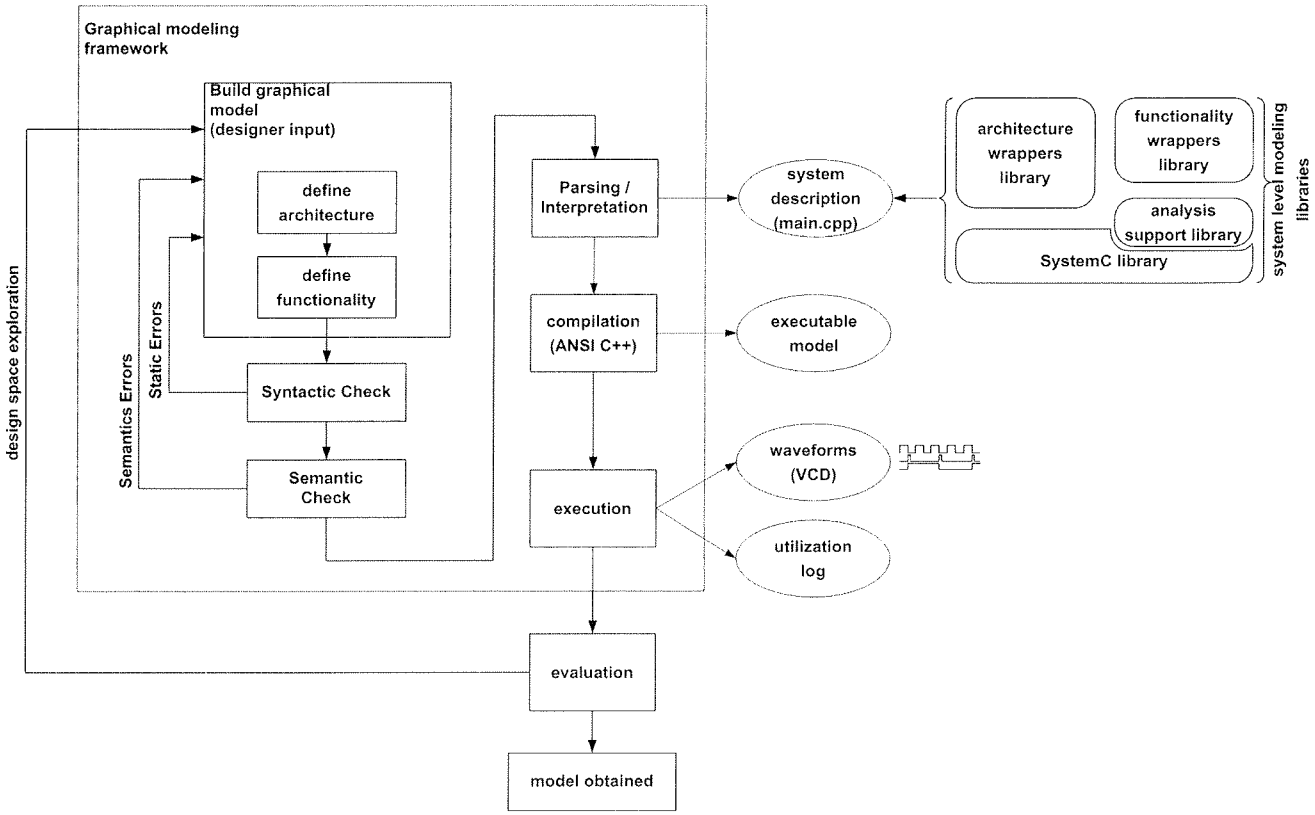

Fig. 2. Graphical framework for design space exploration

process can be divided into four steps:

- In the first step, the designer builds functionality. He/she defines descriptions of each task (time estimations for HW resource usage) and its time dependency (task-start triggers).
- In the second step, the designer determines the system architecture.
- In the third step, the designer specifies mapping of all tasks to appropriate architecture resources.
- In the final step, the designer defines the necessary simulation settings (e.g. simulation step size, max. simulation time, which variables to log into waveforms, and settings for reports of resource utilization).

The design flow continues with compilation of the system model together with wrappers and the SystemC library. An executable model is obtained which produces upon execution waveforms and resource utilization files.

In the next stage, data relevant for further design decisions (e.g. resource utilization, task being idle because of resource contention) are evaluated. Evaluation results are compared with the system specification constraints. If they are not satisfactory, the designer repeats the design cycle with a different system implementation.

During design-space exploration various system implementations can be relatively easy to build and feedback about their evaluations results can be used for finding a path towards a solution best meeting system constraints /8/, /9/.

2.2 Graphical Design Flow

While the system modeling libraries provide a great support for simulation and evaluation, the designer still needs to manually describe the system by means of coding. Consequently, this means that coding has to be changed in each repetition of the design cycle, which is an error-prone process. For designers this still represents a heavy burden for quick and efficient design space exploration.

To simplify and speed-up the process of building a system model and enable its faster exploration we developed a domain-specific graphical modeling framework (GF). The framework enables a graphical creation of a high-level system model and interpretation of the model into the SystemC source code. The system architecture is described by inserting and interconnecting reusable library components in a graphical framework. The system functionality is defined in tasks written in the SystemC and presented as blocks in the graphical framework. Connections are used for a graphical presentation of the tasks time dependency and mapping to architectural resources.

The modeling framework also supports simulation settings, selection of reports and variables being logged during simulation execution and definition of stimulators for simulating external signals that this model is dependent of. The graphical framework performs different syntactic checks during model building and interpretation phase thus minimizing designer errors. In this way it greatly helps designers to build an appropriate model more quickly. Automation of compilation and execution stages is also supported.

3 Graphical modeling framework

Graphical modeling environments are used extensively in different domain-specific areas (e.g. Matlab/Simulink for signal processing).

When a system-model developer decides to switch from the design language to a graphical framework, he/she can take one of the two different approaches; either starts developing a new domain-specific GF from the ground up or using one of the already developed generic graphical frameworks that can be configured for particular domain-specific needs. Each approach has some advantages and disadvantages over the opponent.

The first approach allows the developer to fully control his/her design. As developing such framework is quite expensive, this approach is limited to applications with large potential market. The cost of the second approach is lower and the developer's control over the framework is limited. Only toolsets offered by a selected generic framework can be used. This approach is much more appropriate for applications needing only a small amount of installations.

Open source graphical modeling environments found suitable for us are Eclipse Graphical Editing Framework /10/ and Generic Modeling Environment (GME) /11/. We decided to use GME since it is more mature, offers very good user support through online forum and provides tools for easy integration of the interpreter for translating the graphical model.

3.1 Generic Modeling Environment - GME

The Generic Modeling Environment (GME) /12/ is a configurable toolkit used for creating domain-specific modeling, model analysis, model transformation and program synthesis environments. The configuration is accomplished through meta-models specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic and presentation information regarding the application domain. It defines concepts used to construct models, their relationship, organization and graphical presentation, and rules governing model construction.

The modeling paradigm is created by configuring a meta-model using the GME meta-modeling language. Meta-models are used to automatically generate target domain-specific environment. An interesting aspect of this approach is that the environment itself is used to build meta-models. This top-level environment is called a Meta-metamodel.

The generated domain-specific environment is then used to build domain models that are stored in the model database. They are used to automatically generate applications or to synthesize input to different Commercial Off-The-Shelf (COTS) analysis tools. This process is called model interpretation.

Figure 3 depicts how GME is configured to suit domain-specific modeling environment needs. The role of the meta-

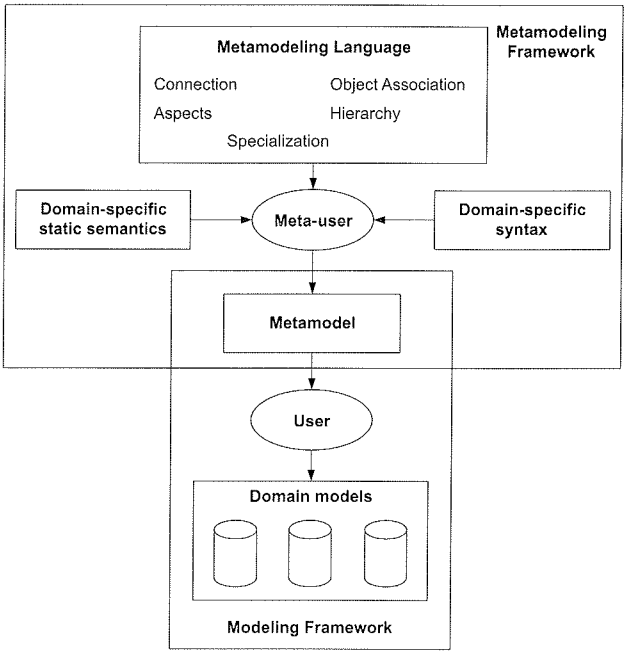

Fig. 3. Configuration of GME needed to obtain a domain-specific modeling framework

user is to construct a domain-specific meta-model with all syntactic, static semantic and presentation information regarding this specific domain. The meta-modeling paradigm is based on the Unified Modeling Language (UML) /14/. The syntactic definitions are modeled using pure UML class diagrams and the static semantics are specified with constraints using the Object Constraint Language (OCL). This process needs to be done just once and the developer of a domain-specific modeling framework takes over the role of a meta-user. Users of this domain-specific framework can build their specific models according to rules defined in the meta-model.

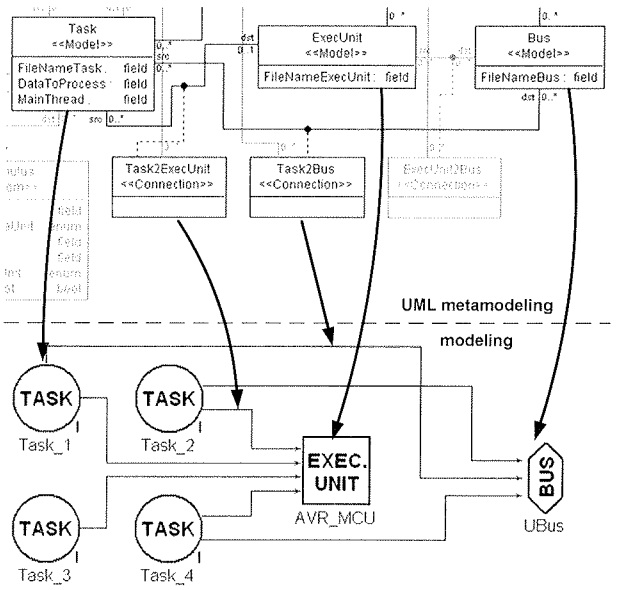

Fig. 4. Creating a domain-specific modeling framework

Figure 4 illustrates a snippet of the UML meta-modeling paradigm and its actual corresponding presentation in GME. The curvy arrows show how individual modeling elements and their relations are defined by different parts of the meta-model.

GME has a built-in set of generic concepts: folders, models, atoms, connections, roles, constraints and aspects. These concepts are the main elements used by the meta-model developer. We will not make a detailed presentation of all of them as this would exceed the scope of this paper. The reader can find it in /12/, /13/. We will just focus on the concept of aspects. Aspects provide visibility control. They are used to allow models to be constructed and observed from different viewpoints. Existence of parts of the domain in a particular aspect is determined by the meta-model. Each part can be either visible or hidden. The concept of aspects allows the user to employ just the parts suited for a selected viewpoint and hide all the others irrelevant for it.

GME also provides high-level C++ and Java interfaces for writing plug-in components to traverse, manipulate and interpret graphical models into an appropriate text description suiting as input to COTS analysis tools. The interpreter needs to be written by the meta-user because interpreter must be able to translate graphical models built according to the meta-model.

3.2 Building paradigm

To configure GME for specific needs of our high-level system modeling, we built a meta-model containing information of all the concepts supported in our system modeling libraries. As mentioned above, the libraries provide wrappers for creating abstract HW resource units (execution and communication units) and wrappers for abstract task creation. Tasks serve for creating a description of algorithm functionality.

Figure 5 shows a part of our meta-model designed by using generic concepts of the GME environment and static UML diagram. For clarity of presentation the only most important concepts of our system-level modeling methodology are presented. The meta-model enables a model of a typical embedded system on a high abstraction level to be made-up as a composition of:

- at least one execution unit (*ExecUnit*),
- any number of communication units (*CommUnit*), and
- at least one task (*Task*).

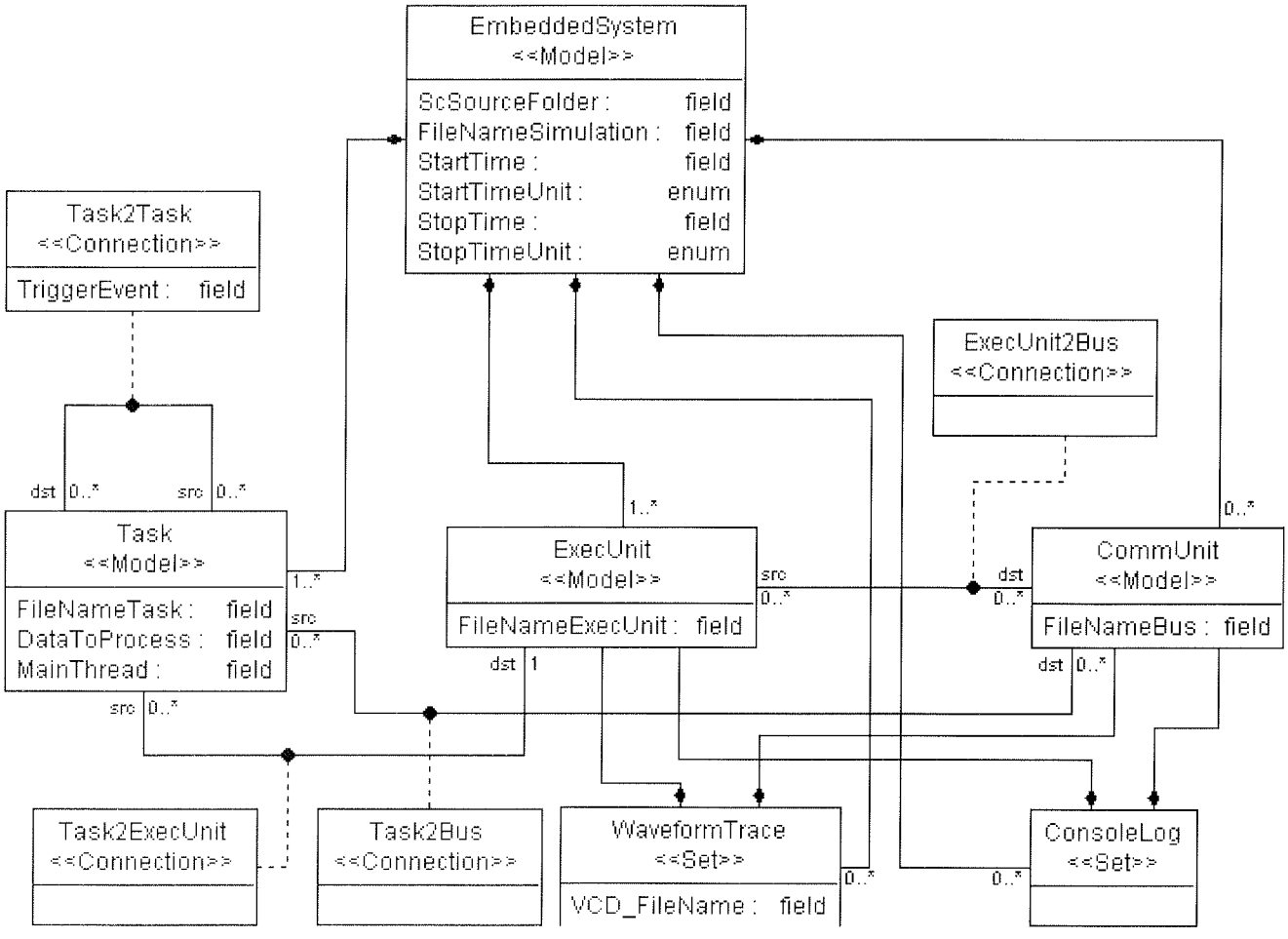

Fig. 5. Snippet of our meta-model

The restrictions for the numbers of instances in the actual model are set by multiplicity constraints (e.g. constraint for the *ExecUnit* is set to: "1..*"). These three units are directly compatible with wrappers in our system modeling libraries. The meta-model also defines possible connections between these elements. The designer can make just the connections permitted in the meta-model. The connections shown in Figure 5 are:

- *ExecUnit2CommUnit*: with these connections the designer defines the communication units available for a selected execution unit. Generally an execution unit can have more than one communication unit and many different execution units can share the same communication units. Instances of execution and communication units connected together compose system architectural resources.
- *Task2Task*: with these connections the designer defines the order of task execution. The order is governed by the tasks' data dependency and the direction from the source to destination has to be followed. Instances of the tasks connected together with the *Task2Task* connections compose system functional description.
- *Task2ExecUnit*: with these connections the designer assigns execution units responsible for execution of a selected task. Each task can be assigned to only one execution unit.
- *Task2CommUnit*: with these connections the designer defines the communication units available for data-manipulation operations of tasks. Generally, a task can use more than one communication unit, but only those available to the assigned execution unit can be used. This means that the designer can select only between those communication units that have been previously attached with *ExecUnit2CommUnit* to the execution unit. Actual constraints for creation of these connections are implemented in a syntactic check performed before starting the model interpretation.

Two blocks representing the GME concepts of *sets* are also shown in Figure 5. The sets are used for selecting and grouping object instances in the system model. With the set *WaveformTrace* the designer defines which HW resources will be traced during simulation. Multiple *WaveformTrace* sets with different members can be used. Each of them represents a different VCD (Value Change Dump) waveform file. A *ConsoleLog* set defines the HW resources used for printing the resource utilization log. This information is gathered and printed after the actual simulation ends.

Besides the presented blocks, the meta-model contains also some other elements required for model construction and simulation setup. All of them are listed in Table 1. The event splitter and event joiner are used for defining the order of task execution. The event joiner performs an addition of multiple input events when starting a specific task depends on execution ending of multiple tasks. Event splitter triggers multiple tasks in a certain order and can be used for modeling a SW scheduler. Start and stop events are used for control of the simulation process. External event-generator elements serve for imitating input signals coming from the surroundings where our system will be operating.

The concept of aspects in GME provides visibility control. The aspects allow models to be constructed and viewed from different viewpoints. They show only elements relevant in a particular aspect. In our meta-model we implemented four different aspects in which a model of an embedded system can be viewed.

- In the task triggering aspect, the designer enters functionality of the system by placing and connecting task instances. The simulation setup elements (start and stop events) and external-event generators are also defined in this aspect.
- In the architecture aspect, instances of hardware resources (execution and communication units) are placed and connections *ExecUnit2CommUnit* are defined.
- The mapping aspect serves for mapping tasks to appropriate hardware resources. Only connections among the already defined instances can be made.
- In the simulation setting aspect, *WaveformTrace* and *ConsoleLog* set elements are instantiated and their appropriate members defined.

Table 1 lists all of the implemented elements of our meta-model in conjunction with the visibility aspects. Even if a specific element is visible in more than one aspect, it can be instantiated or modified only in its primary aspect. The primary aspect is denoted with a shadowed cell.

Visibility \ Aspect	Task Triggering	Architecture	Mapping	Simulation Settings
Task	•		•	
Event Splitter	•			
Event Joiner	•			
Execution Unit		•	•	•
Bus		•	•	•
Start Event	•			
Stop Event	•			
External Event Gen.	•			•
Waveform Trace				•
Console Log (usage)				•

Table 1. Visibility of elements depends on the aspect

For connecting all the elements together, we defined proper connections in the meta-model. As mentioned above, we did not describe all of them since this is not crucial for understanding the idea of our approach. At this point it needs just to be noted that the possibility of making connections also depends on the aspect. All the possible connections implemented in our meta-model and the ability of making them dependent on a particular aspect, are presented in Table 2.

Connection \ Aspect	Task Triggering	Architecture	Mapping
Task2Task	•		
Task2Event Splitter	•		
Task2Event Joiner	•		
EventJoiner2EventSplitter	•		
ExternalEventGenerator2Task	•		
StartEvent2Task	•		
Task2StopEvent	•		
Task2ExecUnit			•
Task2CommUnit			•
ExecUnit2CommUnit		•	

Table 2. Possibility of making connections depends on the aspect

3.3 Model interpretation

Very important part of our graphical modeling framework is the model interpreter. The GME provides high-level C++ and Java interfaces for writing plug-in components to traverse, manipulate and interpret models. The purpose of the interpreter is to translate all information captured in the graphical model into a textual description.

We designed an interpreter which produces a source code description of the system components compatible with the SystemC and our system modeling libraries. The interpreter is capable of handling all the concepts defined in our meta-model. It is written in the C++ and based on the MFC library.

Before an actual interpretation begins, different syntactic and semantic checks are performed to verify the graphical model. Errors are reported and the designer is guided to repair the model. The interpreter generates the SystemC source code together with appropriate project files for automatic compilation and linking. Finally, an executable description of the system model is obtained.

4 Case study

To see how our graphical modeling framework operates in practice, system-level modeling of an existing real-time embedded system will be presented. Since the embedded system is actually already built, its performance can be extracted from the implementation model or measured in the system. Performance estimation before actual implementation was not possible since no modeling framework suitable for heterogeneous system simulation was available at the time. We will show that using our graphical framework for modeling the observed system on a high abstraction level enables performance estimation before the implementation is made. The framework allows very easy exploration of different system implementations.

The case study presents an Illumination and Camera Controller (ICC) /15/ used in computer vision applications for high-speed control of illumination units and triggering line cameras. This is a typical control-oriented embedded system where two position encoders are used for triggering events and computing outputs in real time. A USB communication port is available for setting the operating parameters.

Figure 6 shows a hardware platform for implementation of the ICC. The available hardware resources include an AVR microprocessor, CPLD, USB transceiver, RAM data memory and some other peripheral devices. As this system operates in a time-critical environment, it is crucial to assure that it operates as a hard real-time system. For all these reasons it is reasonable to develop it at the system-level.

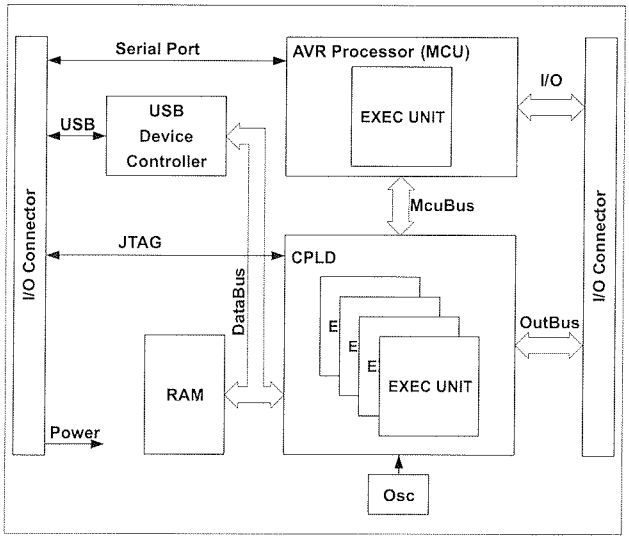

Fig. 6. Platform of illumination and camera controller

4.1 Model construction

Construction of the ICC model on the high abstraction level was performed in four aspects using graphical elements from the meta-model.

In the first aspect, functionality of the ICC is defined. Functionality of the system can be divided into eight different tasks presented in Figure 7. Tasks *T0H* and *T0S* serve for setting the operating parameters after system start-up and for communication with the operator. Task *T0H* performs communication with the USB transceiver and passing of the parameters to task *T0S* which configures the microprocessor. For simulating the parameter setting right after power-up, a start event element *Event_T0* is used.

Operation of the ICC is triggered by two external encoders: absolute position encoder (APE) and incremental position encoder (IPE). We used two external event generator elements *AbsEncIRQ* and *IncEncIRQ* for modeling APE and IPE, respectively. Task *T1* reads IPE events and triggers tasks *T2* and *T3A*. Task *T2* reads new illumination and camera control data from RAM and sends them to the output bus. Since the IPE does not give an absolute position, task *T3A* performs actual re-calculation of the inspected object position based on the data obtained from both encoders. Task *T3B* reads the data from the APE and performs transformation from the Gray to binary code. The

new value of the object position is calculated in task *T3C*. Tasks *T3A* and *T3C* generate output events when a specified position boundary is reached. The events are combined into the *EventJoiner* connected to task *T3D*. This task generates page trigger signals for cameras and resets the illumination control. The output event is in our model connected to the *Event_Stop* element for simulation purposes.

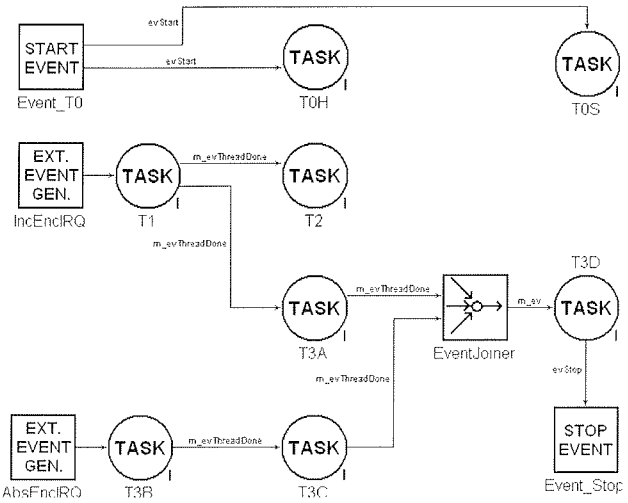

Fig. 7. Functionality description in the task triggering aspect

The possible parallel execution of the tasks is limited by their data dependency and available HW resources. The hardware resources are defined in the architecture aspect by placing instances of execution and communication units. The microprocessor contains only one execution unit capable of executing many different software tasks. On the other hand, the CPLD device can implement more special purpose execution units operating in parallel, but is limited with its size.

In the proposed model, tasks *T0H, T1, T2* and *T3B* are implemented in CPLD. The architecture description contains four CPLD and one AVR execution unit instances, as presented in Figure 8. The ICC platform communication buses *McuBus*, *DataBus* and *OutBus* are also instantiated and connected to appropriate execution units.

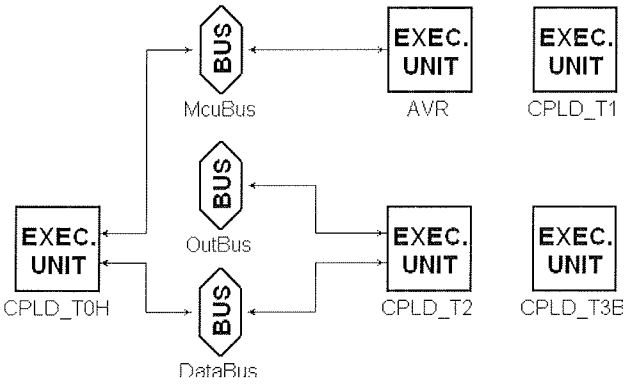

Fig. 8. Architecture of ICC on a high abstraction level

Mapping of the tasks to execution and communication units is actually defined in the mapping aspect, as shown in Figure 9. Each task is mapped to one execution unit and zero or more communication units. The actual use of the communication units is defined in the task description. Tasks *T0S, T3A, T3C* and *T3D* are assigned to execution unit *AVR* representing the microprocessor in the actual ICC system. All other tasks have their own execution units. It should also be mentioned that task *T2* can use two communication units (*DataBus* and *OutBus*).

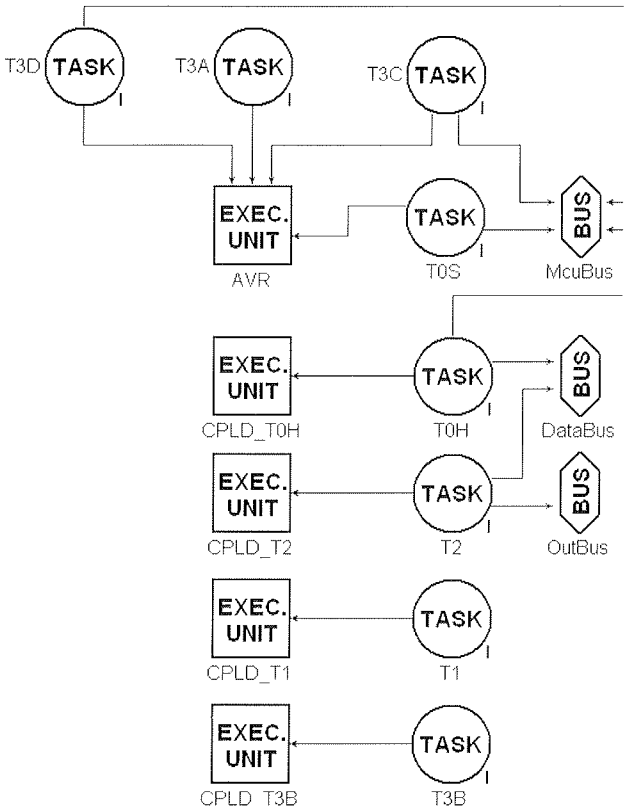

Fig. 9. Mapping tasks on architecture resources

At this point let us briefly explain how the tasks are actually described in SystemC. The task model is defined in terms of architectural resource usage and no actual algorithm is specified. Figure 10 lists a high-level description of task *T2*. This task reads data from the data bus and sends it to the output bus. In order to transfer a block of 8 bytes, the process is repeated eight times.

```
for(int i=0; i<8; i++)
{                  // 8x - for each D/A channel
   m_pExecUnit->GetData(this,4,false, s
        (ICommUnit*) &DataBus,  -1);
                   // gets data from data storage
                   // uses DataBus
                   // need four cycles
  m_pExecUnit->WriteData(this,1,false,
        (ICommUnit*) &OutBus,  -1);
                   // write data to D/A converter
                   // uses OutBus
                   // needs one cycle
}
```

Fig. 10. High-level description of task T2 in SystemC

The predefined methods *GetData* and *WriteData* from our system modeling libraries are used for modeling data transfer. The designer needs to supply the pointer to an appropriate communication unit and estimation of the cycle duration. Both data transfer requests are performed through execution unit interface (*m_pExecUnit*). The libraries also provide methods for modeling and estimation of computational tasks (e.g. *Add*, *Mult*, *Wait*) which can be used for high-level task descriptions.

An image from the simulation setting aspect is shown in Figure 11. A set named *WaveVCD_MCU* is selected and its members (both external-event generators, execution unit *AVR* and communication units *McuBus* and *OutBus)* are shown. All the other architectural resources are shadowed. In this way, the designer defines resources used for producing VCD traces and console log files during model execution.

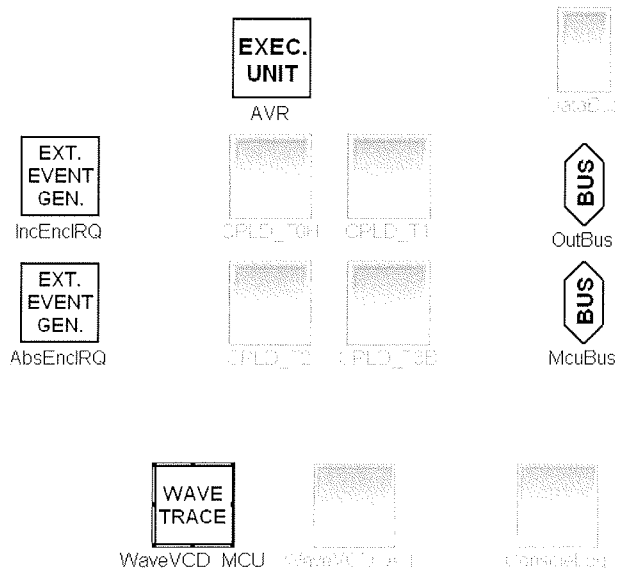

Fig. 11. Selecting hardware resources for waveform traces

4.2 Results

System model evaluation results are obtained after interpretation, compilation and execution of the designed graphical model. Their analysis provides basis for further design decisions.

Table 3 summarizes a part of the utilization log regarding the microprocessor (*AVR*) and the bus *McuBus* that transfers data to the CPLD device. The table is split into the architectural part and functionality part (Tasks). The architectural part shows the resource execution time (RET), i.e. summation of the time the services are required from a specific resource. Complementary, the functionality part presents the task active and wait timings (in % of RET). The numbers stated in the *active* column represent the percentage of the time a specific task is being actively executed on a specific resource. Similarly, the numbers stated in the *wait* column represent the time a specific task has to wait for a specific resource to become available - this is the time interval during which a task may be executed regarding data dependency, but its execution is not started because of the unavailability of HW resources.

Architecture	**AVR**		**McuBus**	
RET[ns]	358 144		134 500	
Tasks	**active**	**wait**	**Active**	**wait**
T0S [% RET]	26.8	0	59.5	0
T3A [% RET]	2.1	5.1	/	/
T3C [% RET]	1.3	0	3.3	0
T3D [% RET]	69.8	0	37.2	0
total [ns]	552 644			

Table 3. Utilization log for AVR and McuBus

Analysis of the results from Table 3 shows that the *AVR* microprocessor is active for about 65% of the simulation time and *McuBus* is active for about 24% of it. Task *T3D* requires most of the processors active time (69.8%) and produces 37.2% of the *McuBus* activity. Waiting can be observed for task T3A (5.1%) caused by the resource contention.

If utilization of a particular resource is not found appropriate, the mapping on architectural resources can be revised. If the real-time constraints are still not met, the functionality description can be revised (e.g. revise algorithm).

The timing diagram, as presented in Figure 12, provides detailed information for the model evaluation. The actual waiting time during resource contention and task execution times can be observed and used for verification of real-time constraints. Under resource contention, task *T3A,* for example, waits 2030ns for an execution unit to become available, but it takes only 84ns to actually execute it.

5 Conclusion and future work

We present a graphical modeling framework used for high-level modeling of heterogeneous systems. It provides graphical design elements for using modeling wrappers from system modeling libraries. Graphical modeling relieves the designer of manual typing the source code and thus hides many details of the SystemC code that normally need to be taken care of. Thus the designer can put more effort on actual modeling. Our graphical framework also provides different constraint checks during modeling and integrates support for simulation settings. When modeling is completed, an executable model is automatically generated to simulate the system behavior on a high abstraction level. Our case study exemplifies the use of our framework and shows information obtained from the executable model built on a high-abstraction level. Evaluation of this information serves as a basis for model evaluation and further design decisions. Graphical modeling enables rapid changes in the model (e.g. changes in the mapping aspect can give

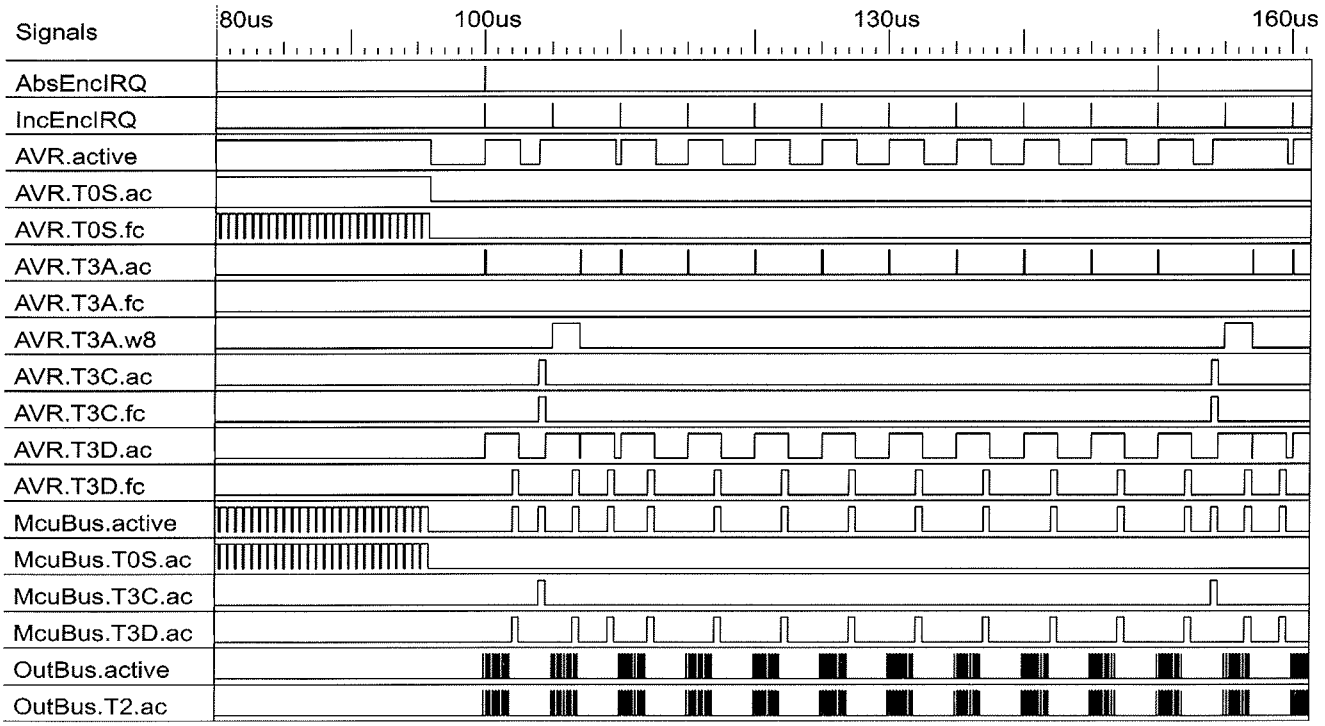

Fig. 12. Timing diagram of selected tasks execution

better results) without time-consuming manual SystemC code rewriting.

In our future work we intend to include support for modeling task interruption and to handle priorities of tasks execution in order to achieve efficient modeling of operating systems. We plan to provide a hierarchical approach to model building as it will significantly improve handling complexity of large models. Our intention is to enable a graphical composition of the task description by providing a library of commonly used methods. Predefined methods for construction of a task high-level description (e.g. *Add*, *Multiply*, *GetData*, *WriteData*) will be thus made available to the designer. We also wish to implement some constraint checks using the OCL language rather than performing them before the interpretation phase. Using OCL, these constraint checks will be performed during the model construction phase.

6 References

/1/ A. A. Jerraya, Long Term Trends for Embedded System Design, CEPA 2 Workshop - Digital Platforms for Defence, Brussels, Belgium, March 15-16, 2005.

/2/ A. A. Jerraya, W. Wolf, Hardware/Software Interface Codesign for Embedded Systems, IEEE Computer Society, vol. 38, no. 2, pp. 63-69, February 2005

/3/ VHDL homepage, http://www.vhdl.org/.

/4/ A. Habibi, S. Tahar: A Survey on System-On-a-Chip Design Languages, Proc. IEEE 3rd International Workshop on System-on-Chip (IWSOC'03), IEEE Computer Society Press, pp. 212-215, June-July 2003

/5/ L. Cai, D. Gajski. Transaction Level Modeling: An Overview. CODES+ISSS'03, October 2003, Newport Beach, California, USA, Pp. 19-24, ISBN:1-58113-742-7

/6/ SystemC OSCI homepage: http://www.systemc.org/

/7/ D. C. Black & J. Donovan. SystemC from the ground up. Springer, ISBN: 1402079885, June 2004

/8/ J. Dedič: Enovito razvojno okolje za sočasno načrtovanje strojne in programske opreme, doktorska disertacija, Univerza v Ljubljani, Fakulteta za elektrotehniko, 2006

/9/ J. Dedič, M. Finc, A Trost: A Framework For High-Level System Design Exploration, Informacije MIDEM, vol. 36, no. 3(119) , pp. 151-160, 2006

/10/ Eclipse GEF project homepage: http://www.eclipse.org/gef/

/11/ GME project homepage: http://www.isis.vanderbilt.edu/Projects/gme

/12/ A. Ledeczi, et al. The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

/13/ Institute for Software Integrated Systems, Vanderbilt University, A Generic Modeling Environment: GME 6 User's Manual, Version 6.0

/14/ OMG UML homepage: http://www.uml.org/

/15/ A. Trost, B Likar, Embedded Development Platform and Applications, Proc. 39th MIDEM Conference, pp. 255-260, October 2003

Klemen Perko, B.Sc.
Sipronika d.o.o., Tržaška 2, 1000 Ljubljana
klemen.perko@sipronika.si

Assistant Prof., Dr. Andrej Trost,
University of Ljubljana, Faculty of Electrical Engineering,
Tržaška 25, 1000 Ljubljana, Slovenia

Prispelo (Arrived): 06.03.2007 *Sprejeto (Accepted): 15.09.2007*