

Model specification and visualization in the MASSIVE project

Laurențiu Nicolae Ed Deprettere Bart Kienhuis

Leiden Institute for Advanced Computer Science - Leiden University, Leiden, The Netherlands

E-mail: {nld,edd,kienhuis}@liacs.nl

Abstract— The MASSIVE project aims at developing methods and tools for the exploring of large-scale embedded system designs at a high level of abstraction. One of the key problems that must be addressed is mastering the complexity of both systems and tools. For effective and efficient exploration of such system designs, one needs models of behavior and structure of applications and architectures and a fast simulator for measuring the performance of the combination of both. It turns out that existing frameworks have either good modeling capabilities or good simulator capabilities, but not both at the same time. In the MASSIVE project we have investigated the possibility of interfacing the Ptolemy framework from Berkeley, and the System Simulation Framework from Dartmouth (DaSSF). Ptolemy is used to create, visualize and manipulate structural models. DaSSF is used for fast clustered simulation of system level designs. Structural models are exported from Ptolemy to DaSSF and performance numbers obtained from the simulator are imported back to Ptolemy for further evaluation and visualization. We have implemented the interfacing of these two frameworks and evaluated the usefulness of the new environment by working out an illustrative exploration example.

Keywords— embedded systems design, design space exploration, system level modeling

I. INTRODUCTION

The MASSIVE project is a research project that is focused on the development of a methodology for high level design of large scale embedded digital signal processing systems. As carrier for the project, a concrete case is planned to be tackled: exploring alternative system level implementations of the digital signal processing part of a Square Kilometer Array (SKA) type radio telescope [1], [2], consisting of 10,000,000 RF sensors, demanding multiple Terra-operations per second of processing power.

The problem we are addressing is design space exploration for such systems, using models for both the possible architectures and for the applications that run on these architectures. After deciding upon one model from each category, the designer chooses a mapping of the application onto the architecture and executes a simulation of the resulting system model. Af-

ter the simulation is over, the performance numbers from the simulation are sent back to the designer, who decides whether he should explore other alternatives.

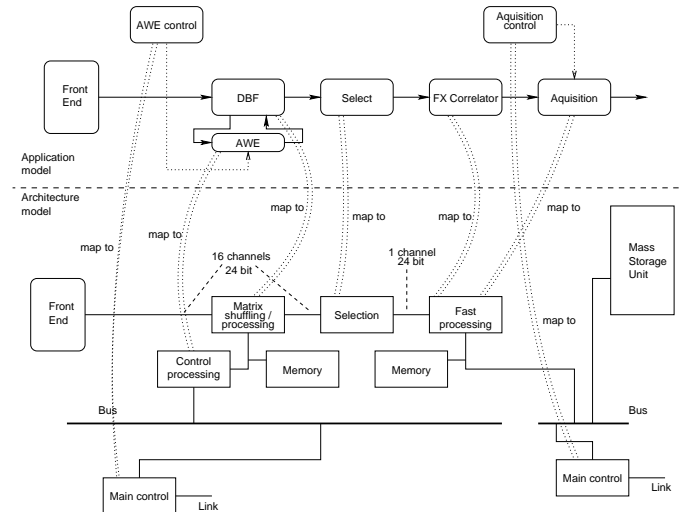


Fig. 1. The THEA structure

In order to understand better the type of systems we have in mind with this approach, we present the model of the digital signal processing part of the THousand Elements Array or THEA [3] (see Figure 1), which is one of the demonstrator systems that was built at ASTRON [4] as part of the SKA development program. In the upper part of the picture is the application model, described as a network of processes that run in parallel and communicate to each other through unbounded FIFO channels, known also as a Kahn Process Network (KPN) [5]. This model describes the behavior of a particular system. The names in the boxes represent algorithms such as Digital Beam-Forming (DBF) or Adaptive Weight Estimation (AWE).

On the lower part of the system we can see the architecture model, which is built using computational blocks such as FPGAs, DSPs or industrial PCs, linked with different types of connections, such as Ethernet or high-speed optical links. Each of the algorithms from the behavioral model are mapped on one of the computational blocks. For instance, the DBF algorithm is mapped in THEA on a block called Matrix

Shuffling / Processing (MSP), consisting of 6 inter-linked FPGAs, and the AWE algorithm is run on an industrial PC, linked with the MSP block via an Ethernet connection. The paradigm allows also the mapping of more algorithms on a single block.

This model is kept on a high level of abstraction, allowing the designer to quickly explore different architectures and mapping alternatives or different applications. Another issue here is the exploration of individual blocks in both sides of the system model using the same methods, in order to derive block parameters in the larger model. This procedure would allow the designer to get more accurate results.

It is obvious that for the exploration of such systems we need both modeling and simulating tools. We soon discovered that the tools available today in the community are not designed for this type of problems, but there are some that can be extended beyond their original purpose. However, they usually concentrate either on the modeling capabilities or on the performance of their simulator. Bearing these facts in mind, we have decided to separate the modeling and the simulation altogether by using different tools for these tasks, namely the Ptolemy environment from Berkeley and the DaSSF simulator from Dartmouth, for which we have implemented and evaluated an interface prototype. The results of this evaluation are promising, and we are confident that we will be able to use this framework in the context of the MASSIVE project.

The remaining of the paper is organized as follows: we will first briefly present in Section II the tools we decide to use, then we will focus in Section III on the modeling part of our framework. In Section IV we will talk about methods to get feedback information from the simulator side and conclude in Section V with an example we have investigated in order to evaluate the new environment.

II. MODELING AND SIMULATING

After evaluating several simulators, we discovered that the closest tool to satisfy our needs was DaSSF[6], a C++ implementation of the Scalable Simulation Framework[7] built in Dartmouth College, USA. DaSSF is a highly efficient event-based simulator, capable of running large-scale simulations on parallel systems; for a presentation of DaSSF within the MASSIVE context and the motives behind our decision, please refer to [8]. This tool was designed for simulating large networks and we felt confident that it could suit our needs as well. However, we

soon discovered that the modeling side of this tool did not support some key features that we decided that were needed in our framework, features such as a good graphical user interface that would permit the designer to create, evaluate and modify models in a easy and intuitive fashion.

Therefore, we decided that we needed another modeling environment, one that would support hierarchy, would be scalable and would be capable to export both the application and the architecture model in a way that would allow us to translate it to DaSSF. We have finally decided to use the Ptolemy[9] environment from Berkeley University of California. This environment was used successfully in other projects within our group, and the positive results compelled us to go on with testing it in the MASSIVE context.

The powerful graphical user interface of Ptolemy allows the designer to manipulate easily the structure of models, by using library building blocks and validating the interconnections between these blocks (see below for a discussion about the behavior of the models). The environment provides a textual specification of the models using a subset of the XML language. This language supports hierarchy by default, and this feature has been exploited by the creators of Vergil, the graphical user interface included in the Ptolemy suite. Another plus is the portability of the Ptolemy package, emerging from the fact that it is implemented in Java.

There are a few issues that need to be addressed here. The first question is whether to specify the behavior of the model in Ptolemy and export it to DaSSF, or to specify this behavior in DaSSF only. We did not see any advantage in writing behavioral code in Java within the Ptolemy environment and then convert it to C++ and export it to DaSSF. Therefore, we decided that we will separate the structure of the model, which is easier to create and maintain in Ptolemy, from the actual behavior, which is to be kept entirely on the simulator side. The actors used in the Ptolemy models are not stand-alone; they rely on the functionality provided by their counterparts from DaSSF. Bearing this facts in mind, it was obvious that in order to create a bridge between Ptolemy and DaSSF, we needed to communicate the structural information of the models.

Another issue that must be addressed here is how to gather informations from the simulation and present these informations to the user. We decided to use for this purpose the Ptolemy built-in visualization and exploration features, which can be used within Vergil,

stand-alone and even exported into a Java applet that can be posted on a Web page.

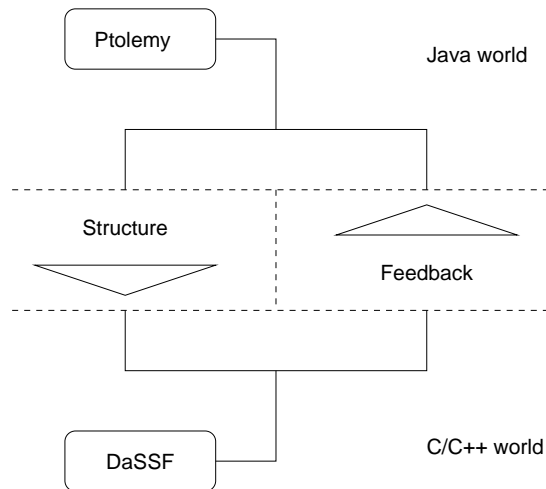


Fig. 2. A view of the system

As seen in Figure 2, we have two main problems that we need to solve. In the following two sections we will discuss how we tackled these problems and what have been the steps we took towards our goal.

III. THE MODELING ENVIRONMENT

One of the most important reasons behind our decision to use the Ptolemy environment for creating the structural model of our application and architecture is the language that is used for the specification of such a model. Called MOdeling Markup Language (or MoML, for short), it is a subset of XML, specially designed for specifying interconnections of parametrized components. This powerful language supports both hierarchy and the use of library blocks, making it easy for the designer to create complex models.

We have depicted in Figure 3 the classical Producer-Consumer example, using a Kahn Process Network (KPN) (see Section I) for the application description and standard SSF blocks for the architecture model. The application in this example (upper window in Figure 3) is very simple; we have two processes connected by an unbounded FIFO channel. These processes correspond to the two entities bearing the same name from the architecture model (lower window in Figure 3); the communication between entities is realized using a bus and a memory.

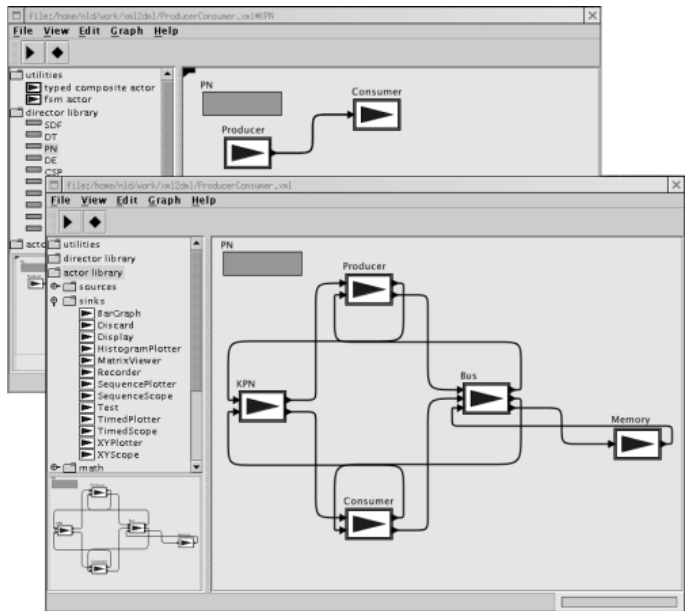


Fig. 3. Vergil - the Producer-Consumer example

The interesting thing here, however, is that the whole application model is contained within a special block in the architecture model, a block called KPN. The reason for this decision is that the application and the architecture models are cosimulated in the DaSSF side. The application contains the functionality of the model and drives the architecture via messages sent through mapping channels (also visible in the lower window in Figure 3), whereas the architecture only implements the timing behavior of the system.

The Ptolemy actors used to create this model can be added to Vergil by specifying their structural part and including it into the Vergil actor library. An actor must contain:

- The definition of actor ports. Here we specify the name, type - input, output or multiport - and data type for the ports. The GUI uses the type specified here in order to validate the connexions between two actor ports.
- Parameters' definition. Here there can be specified all the relevant parameters of a particular actor. Parameters need to have types and can be initialized directly inside the class. Both ports and parameters can be dynamically added, removed or modified in Vergil.

Our intention is to have library blocks already implemented in DaSSF, with their structural equivalent in the actor library of Ptolemy. In this way we can allow the designer to just draw and execute the model from within Vergil.

The DaSSF simulator uses the Domain Modeling Language (DML) [10] for describing the structure of

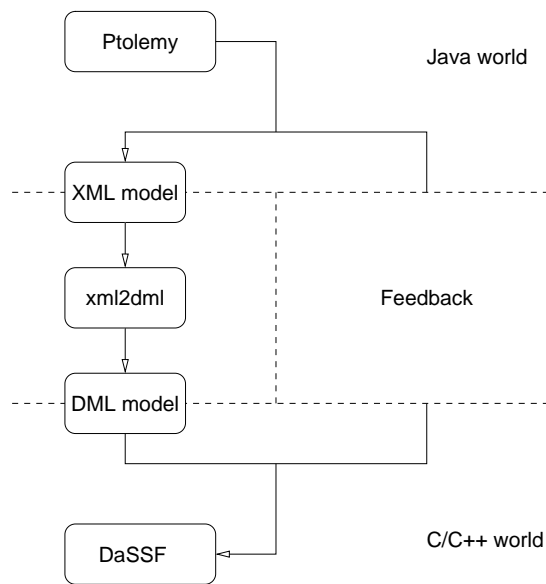


Fig. 4. Closing the structure path

models. A DML script could be described as a recursively defined list of attributes, similar in a way with the structure of a database. An example of such a file can be found in Appendix A. A DML model is very hard to create and debug when we are dealing with models of increased complexity. Due to the fact that the description of an entity is broken in two - properties and connexions - mistakes are easy to make, even in smaller models.

However, by using the Vergil editor, the designer has access to a set of features that simplify the way he creates the model:

- **Drag-and-drop editing** - the designer can add a new actor just by dragging it from the library on the left and dropping it on the canvas;
- **Copy-paste support** - when the designer executes a copy-paste, the entire underlying MoML code associated with a particular actor is duplicated in the model description file;
- **Connection validation** - for every connection drawn between two blocks the environment verifies if the types of the two interconnected ports are compatible.

In order to close the structural modeling path, we have created a program that converts the MoML textual description file into a DML script (see Figure 4). The program, named *xml2dml*, is a command-line utility that takes in the file from Ptolemy and, after extracting the relevant informations, creates a human-readable DML file. As future work, we intend to integrate this into Vergil, allowing the designer to see how the changes he makes on the canvas reflect

into the DML file, thus facilitating the better understanding of the model.

IV. GETTING FEEDBACK INFORMATION

We can use Ptolemy to process and visualize information from the simulation; either the final results - which is relatively simple, we can just save the relevant data to a file - or, more important, “live” intermediary results collected during the simulation. For this purpose we can use some special categories of Ptolemy actors (as shown in Figure 5):

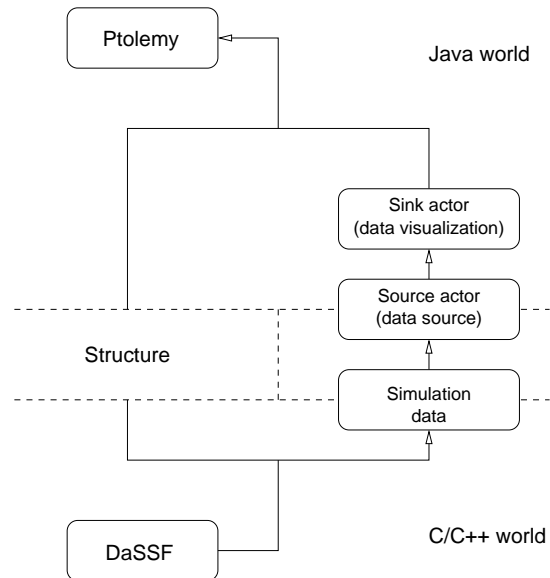


Fig. 5. Closing the feedback path with Sources and Sinks

- **Sources:** these actors can “create” data sets that can be read into other actors. They can contain a generator function or - better in our case - a file reader that we can use to read either from a results file created by the simulator or from an UNIX pipe, in order to create the “live” data feed into Ptolemy.
- **Sinks:** these actors employ various visualization techniques to display a set of data read from a channel.

Due to the particularities of the system, the feedback loop has to be implemented separately from the actual models, allowing a designer to create, for instance, different views of the system according to his needs. These views can be run either within Vergil or stand-alone. The Ptolemy system offers to the designer also the possibility to export these views as Java applets that can be used in Web pages.

In the Ptolemy environment there are already a number of general-purpose sinks that can be used by the designer for the visualization of his simulation

data, sinks like HistogramPlotter or MatrixViewer. If these do not suffice, one can always implement his own dedicated sink, by following the actor design guidelines from the Ptolemy II manual.

We are currently working on implementing this feedback loop into our environment.

V. THE M-JPEG EXAMPLE

In order to verify the validity of our approach, we have taken a problem already investigated within our group, namely the M-JPEG algorithm implementation[11]. The M-JPEG is a picture sequence compression algorithm that uses a JPEG-based compression technique to each picture from the sequence[12]. We have chosen this example because it was modeled in a similar way with the one we have to design, and the algorithm was already partitioned and tested in the Spade[13] tool.

You can see the application model of computation partitioned as a KPN in Figure 6. The KPN's unbounded FIFOs are represented by the channels, and the KPN's processes are represented by the Ptolemy actors. These actors have no functionality, for this is implemented on the simulator's side; they merely contain the parameters and ports deduced from the structure of the application.

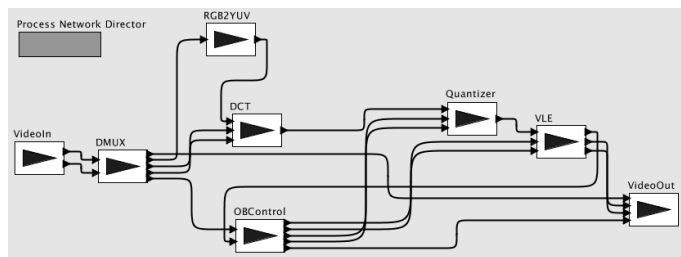


Fig. 6. MJPEG: the application, specified as a KPN

We have also created in Ptolemy the architecture model on which this application was mapped (see Figure 7). The corresponding KPN, represented here by a single block on the left of the picture, is connected to the architecture by mapping channels (see Section III). Each of these channels is connected to one of the application nodes, their role being to inform the simulator which process runs on which architecture block.

The architecture model contains a microprocessor, two DSPs and two I/O ports, which are modeled as *Entities*, a *Memory* and a *Bus*. We have each entity connected to the bus, which is in turn connected to the memory. As you may notice, there are also some

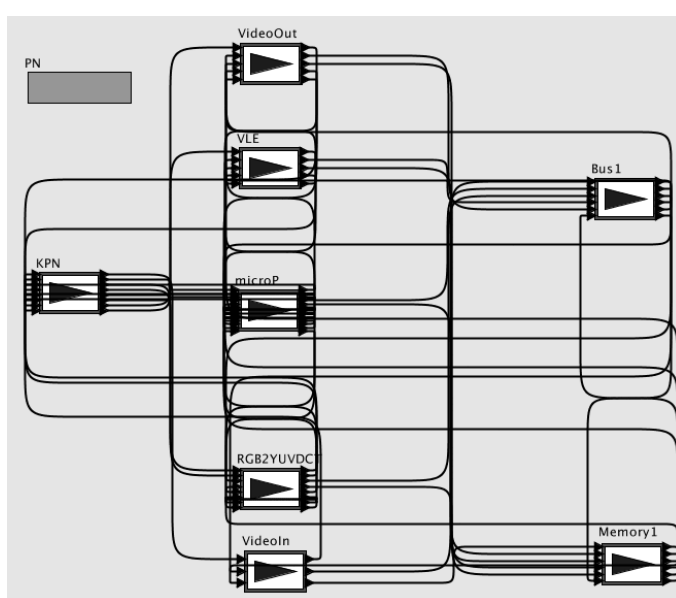


Fig. 7. MJPEG: the proposed architecture

blocks which require direct memory access, and are connected accordingly.

The resulting model is processed through *xml2dml* and sent towards the simulator, which performs a cosimulation of the MJPEG algorithm described by the KPN and the specified architecture. As you can see, we have completely separated the structure information, now described in the Ptolemy environment, and the functionality, which is implemented in DaSSF.

Unfortunately Vergil does not include yet some features like “snap to grid”, and the channels are drawn automatically on the canvas by the system. That is why the architecture model looks cluttered; however, this fact by no means diminishes the power of the tool, and the designers promise to correct these details in the future releases. We can also take further steps towards simplifying the model, by considering for instance that some of the channels are implicit and adding them during the conversion to DML.

VI. CONCLUSIONS

In this paper, we presented model specification and visualization methods suited for large-scale embedded systems design. We have successfully separated the modeling and the simulating environments, thus being able to use for each of these tasks the tools that best suited our needs. We are now able to offer the designer an interface which supports library blocks and a hierarchy.

As future work, we intend to further develop the *xml2dml* in order to accommodate the future version of DML currently under development in Dart-

mouth. Furthermore, we plan to integrate *xml2dml* into Ptolemy, in order to reflect each change the designer does on the canvas into the DML model file. We will also implement the feedback loop by processing and visualizing the simulation data within Ptolemy.

REFERENCES

- [1] "SKA Home Page," <http://www.astron.nl/ska>.
- [2] C.J. Lonsdale and R.J. Cappallo, "Concepts for a Large-N SKA," in *Proc. of Int. Symposium on Technologies for Large Antenna Arrays*, Dwingeloo, The Netherlands, Apr.12-14 1999.
- [3] B. Smolders and D. Kant, "System Specification for THEA," Tech. Rep. SKA-development-140, NFRA, 1999.
- [4] "ASTRON / NFRA Home Page," <http://www.astron.nl>.
- [5] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*, 1974, North-Holland Publishing Co.
- [6] "The DaSSF project," <http://www.cs.dartmouth.edu/jasonliu/projects/ssf/>.
- [7] "SSF Research Network," <http://www.ssfnet.org>.
- [8] F. Sluiter, "Parallel systems simulation in the MASSIVE project," in *Proc. 2001 Workshop on Embedded Systems (PROGRESS'01)*, Valthoven, The Netherlands, Oct. 2001.
- [9] "The Ptolemy project," <http://ptolemy.eecs.berkeley.edu/>.
- [10] "Domain Modeling Language (DML) Reference Manual," <http://www.ssfnet.org/SSFdocs/dmlReference.html>.
- [11] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with spade: an m-jpeg case study," in *Int. Conference on Computer Aided Design (IC-CAD'01)*, San Jose CA, USA, Nov. 4-8 2001.
- [12] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards; Algorithms and Architectures*, Kluwer Academic Publishers, 1995.
- [13] P. Lieverse, P. van der Wolf, E. Deprettere, and Kees V., "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems," in *Proc. 1999 Workshop on Signal Processing Systems (SiPS'99)*, Taipei, Taiwan, Oct. 20-22 1999, pp. 181-190.

APPENDIX A: PRODUCERCONSUMER.DML

Here is presented the DML file that was generated from the Producer-Consumer example presented in Section III using *xml2dml*.

```

### DML : ProducerConsumer
### generated by xml2dml (c) 2001 LIACS, Leiden University, All rights reserved
### author Laurentiu Nicolae - nld@liacs.nl

MODEL [ # Beginning of model
  #Entity KPN
  ENTITY [ID 0 INSTANCEOF "KahnProcessNetwork"
    PARAMS [
      STRING %1
      INT %n
    ]
  ]
  #Entity Producer
  ENTITY [ID 1 INSTANCEOF "Entity'ARCH"
    PARAMS [
      STRING "Producer" INT %n # Name, ID
      INT 1 # Traces
      INT 1 # No. of commands
      INT 2 # No. of command parameters
      INT 6 # No. of lines in header
      # List of commands
      STRING "Write'Producer'out'Info" INT 672 INT 0
    ]
  ]
  #Entity Consumer
  ENTITY [ID 2 INSTANCEOF "Entity'ARCH"
    PARAMS [
      STRING "Consumer" INT %n # Name, ID
      INT 1 # Traces
      INT 1 # No. of commands
      INT 2 # No. of command parameters
      INT 6 # No. of lines in header
      # List of commands
      STRING "Read'Consumer'in'Info" INT 672 INT 0
    ]
  ]
  #Entity Bus
  ENTITY [ID 3 INSTANCEOF "Entity'Bus"
    PARAMS [
      INT 8 # Width of bus
      INT 2 # Number of interfaces
    ]
  ]
  #Entity Memory
  ENTITY [ID 4 INSTANCEOF "Entity'Memory"
    PARAMS [
      STRING "Memory" INT %n # Name, ID
      INT 3 # No. of blocks attached to data channel (bus)
      INT 1 # No. of memoryplaces
      INT 2 # No. of command parameters
      INT 6 # No. of lines in header
      # List of commands
      STRING "Write'Producer'out'Info" STRING
      "Read'Consumer'in'Info" INT 672 INT 5
    ]
  ]
  ]
  MAP [FROM 0(TRACE0'OUT) TO 1(TraceIn0) "DELAY 0.0001"]
  MAP [FROM 0(TRACE1'OUT) TO 2(TraceIn0) "DELAY 0.0001"]
  MAP [FROM 1(TraceRequest0) TO 0(TRACE0'REQUEST) "DE-
  LAY 0.0001"]
  MAP [FROM 2(TraceRequest0) TO 0(TRACE1'REQUEST) "DE-
  LAY 0.0001"]
  MAP [FROM 3(DataOut0) TO 1(BusDataIn) "DELAY 0.5"]
  MAP [FROM 1(BusDataOut) TO 3(DataIn0) "DELAY 0.5"]
  MAP [FROM 3(DataOut1) TO 2(BusDataIn) "DELAY 0.5"]
  MAP [FROM 2(BusDataOut) TO 3(DataIn1) "DELAY 0.5"]
  MAP [FROM 4(MemDataOut) TO 3(DataIn2) "DELAY 1.0"]
  MAP [FROM 3(DataOut2) TO 4(MemDataIn) "DELAY 1.0"]
] # End of model

```