System Definition and ESE Data Structure

Lochi Yu, Samar Abdi, Daniel Gajski

Technical Report CECS-09-04
Mar. 25, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

lochi.yu@uci.edu, sabdi@uci.edu, gajski@uci.edu

Abstract

This report describes the Embedded Systems Environment Data Structure (ESE DS) which captures the details of a model of an embedded system, including application and platform information. A platform includes Processing Elements (PEs), shared busses, transducers, channels and routes. A designer can use a GUI to produce this ESEDS and can freely manipulate the platform for design exploration purposes. The output can be used to either produce a Transaction Level Model or a Pin Cycle Accurate Model of the system. To test this data structure, we used 3 different multimedia applications, mapped them into platforms, and applied different operations and transformations. Also, TLMs were generated from this ESEDS and their performance was recorded. We concluded that this data structure is flexible enough for platform transformations and yields enough information to describe an embedded system to generate refined models from it.

Contents

1 Introduction **1**

2 Transaction Level Models **2**

3 Related works **2**

4 System Definition **3**
4.1 Application 3
4.2 Platform 5

5 Embedded Systems Environment Data Structure **7**
5.1 ESEDS Formalism 8
5.1.1 Design 9
5.1.2 Processing Element (PE) 9
5.1.3 Bus 10
5.1.4 Transducer (TX) 11
5.1.5 Communication Channels 12
5.1.6 Route 13
5.1.7 Connection 13
5.2 ESEDS Rules 13

6 Platform Manipulations **14**
6.1 Basic platform operations 14
6.2 Moving processes 14
6.3 Application plus mapping specification 16
6.4 ESE Object configurations 16

7 Experimental Results **18**

8 Conclusions **20**

9 Acknowledgments **23**

References **23**

List of Figures

1 Embedded Systems Environment design flow 4
2 ESE system definition 4
3 Partitioning processes 5
4 Platform definition 6
5 Process mapping 6
6 Channel mapping 7
7 ESEDS element hierarchy 8
8 ESE DS project directory structure 15
9 Routing options for a channel 17
10 JPEG decoder platform 18
11 MP3 decoder platform 19
12 H264 decoder platform 19
13 Basic ESE Operations 20
14 Platform transformations and mapping 21
15 Functional TLM Generation 21
16 Timed TLM Generation 22
17 Execution speedup after platform transformations 22

System Definition and ESE Data Structure

L. Yu, S. Abdi, D.Gajski
Center for Embedded Computer Systems
University of California, Irvine

Mar. 25, 2009

1 Introduction

Embedded systems' rising complexity has forced the industry to adopt new design techniques. Higher engineering costs, shorter time-to-market timeframes and longer design and verification phases demand a change in the traditional design approaches.

In the traditional approach, the *Top-down* methodology, the starting point is an abstract model of the system. This model is further refined in steps, adding more implementation-specific features, until a point where its components can be instantiated from a library.

A second approach is a *Bottom-up* methodology, where the design starts with a set of presynthesized components and builds subsystems with them, up until a whole system has been defined.

Instead of using either a *top-down* or a *bottom-up* approach, the alternative is *Platform-based design* [9, 13]. It adapts the best features of the two approaches in system design. The starting point is not one abstract model of the entire system, but an instance of a *platform*. A platform is a valid composition of elements and their interconnections [5]. With these elements, subsequent refinements will bring the design down to the implementation level. The advantage of this approach is that several platforms are already prebuilt to have an optimized performance for specific type of applications such as multimedia, automotive, or industrial. They can be regarded as a common hardware denominator among those kinds of applications.

The components of the platform can be embedded processors, memories, hardware IPs, Input/Output components or communication elements such as bridges, transducers, or shared busses. For instance, a multimedia platform already has integration of embedded processors, memories, analog-to-digital and digital-to-analog converters, direct memory access module, and specialized hardware for mathematical operations.

Starting with a platform, the designers must map their applications to the processing cores present, and can also consider substituting software components for faster IP cores. In addition, the designers may wish to alter the platform to optimize performance.

In order to capture the whole embedded system, platform details and application details must be included in a model. We propose a data structure based on XML[1] called ESE Data Structure (ESEDS) which solves these problems and also opens up more posibilities of manipulation to a embedded system model. Since it is based on XML, it is extensible to allow future additions to the specification.

Once we have a well-defined data structure representing an embedded system, we can use it in many ways. We can use it to automatically generate[15] an executable model of the system that reflects platform choices: a transaction level model (TLM) of the design. In addition to that, another benefit of having this data structure is that we can freely manipulate the platform to add/remove hardware units or add/remove/move processes from programmable processors. Since we can also produce a timed TLM of the system [8], these system features can be easily modified to achieve the optimum performance.

2 Transaction Level Models

In recent years, transaction level models (TLMs) have emerged as the new paradigm for system design. In transaction level models, the details of communication are separated from the details of computation. Communication between computation modules is modeled as complete transactions (hence the name of the abstraction level), instead of the hundreds of toggling signals in a Bus Functional Model (BFM). While this significantly increases simulation speed, it decreases its accuracy.

We use channels to model the communication between modules, which are a repository of communication services. They provide interfaces that service transaction requests by the computation elements. More communication details can be added in different steps of the process, the same as the computation details. This will increase the accuracy of the computation or communication component. Depending on these factors, we can have different TLMs. In [4] they provide a clear taxonomy of TLMs, based on the accuracy mentioned above.

3 Related works

Several languages have been developed over the years to describe either hardware or software. For instance, Architecture Design Languages (ADL) are used to describe software and/or system architecures. It has been used frequently to describe a specialized processor and synthesize simulators and compilers for it. A comprehensive survey of ADLs has been done in [12]. Some design languages capture the system in a Register Transfer Level, such as UDL/I [3] and MIMOLA[10]. The latter one is used not only for simulation, test generation and code generation, but also for hardware synthesis.

Other groups have also developed languages based on XML, such as XADL[7] is designed to describe software architectures, and its main features are extensibility and flexibility.

To model entire embedded sytems, the language EADL[11] was created to capture both hardware and software components and their interactions. It is meant to facilitate design space exploration and scalable hardware-software co-verification. This language differs from our approach in the following ways: it abstracts all types of platform elements into three kinds of components: software, hardware and bridge components. These components abstract all the processors, busses and embedded OS of the platform. Another difference is that the designer must code in EADL to alter the platform and to do any manipulations on it.

The Dalton Project [14] allows the designer to tune the platform specifications to optimize the system performance, but it is oriented mainly for power consumption, and does not allow platform transformations.

In our approach, the user does not have to deal with the complexities of any code or XML structure. Our frontend GUI eases the design process so that the designer can concentrate con transforming and remapping the platform to achieve optimum performance.

This report is structured in the following way: in Section 4 we define the elements of our system, in Section 5 we describe the data structure used in our tools, including the formalism and rules for ESEDS. In Section 6 we present the platform manipulations that our tools allows and in Section 7, we show how our tools perform when manipulating 3 different multimedia platforms. Finally we present the conclusions in Section 8.

4 System Definition

We have implemented the TLM based design into a tool named *Embedded Systems Environment* or ESE [6]. The entire ESE design flow is shown in Figure 1. Our ESE Frontend has a GUI where the designer can drag and drop components to create the platform. This frontend will manage the XML data structure to insulate the users from the complexity of ESEDS and its internal structure. This report will describe the ESE Data Structure, while the TLM Generator, TLM Estimator and Backend synthesis tool will not be discussed here.

Our system will be defined with two components (see Figure 2):

1. Application

2. Platform

4.1 Application

The application is the component which the designer wishes to run on the embedded system. It may be composed of one or several programs which will run on programmable embedded processors and other specialized function which will either also run on embedded processors or synthesized into specialized hardware units. The application can be composed of one or more processes.

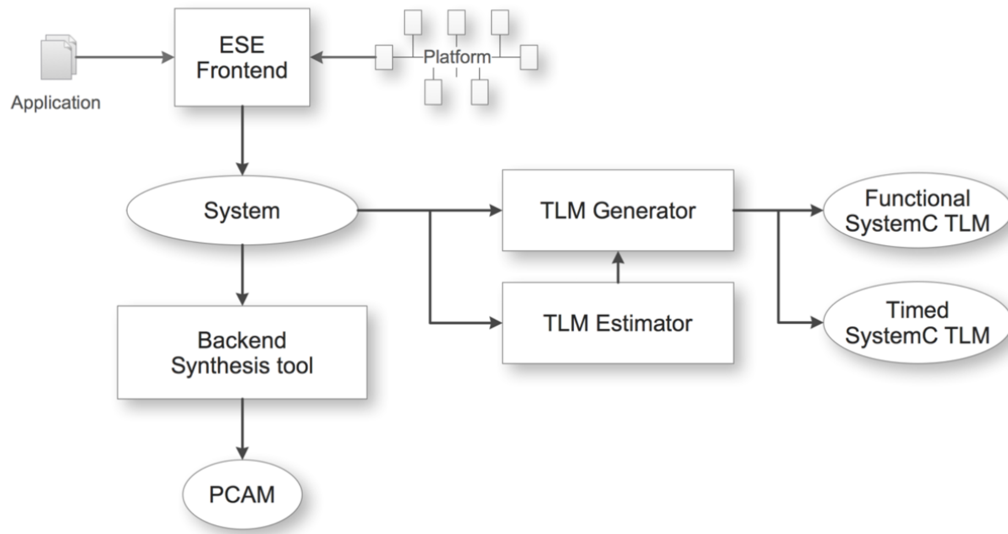

Figure 1: Embedded Systems Environment design flow

Figure 2: ESE system definition

4.2 Platform

The platform is composed of processors, memories, or Hardware IPs, as mentioned above in [5].

Our system model can be viewed as a combination of a behavioral specification that defines its functionality and a set of design decisions to implement the desired functionality. As for the specification part, our tool does not need the C code itself, and only needs the information at the process level. In order to specify the system, there are several steps to be taken:

1. *Partition the application*: the application should be partioned into several processes, separating the main process (at least one) and the specialized functions that should not reside in the same processor. This process is illustrated in Figure 3.

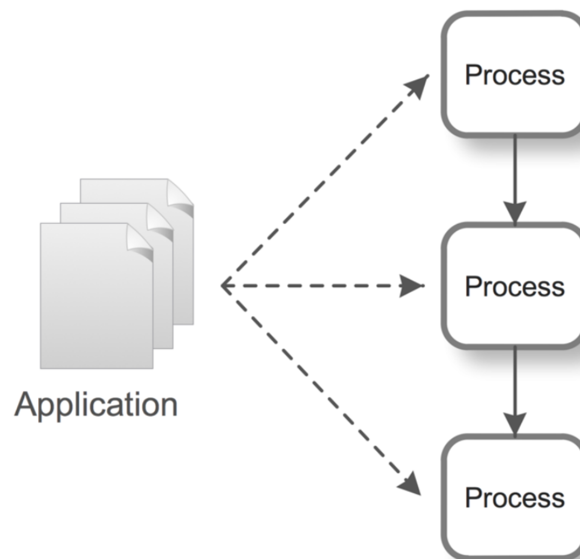

Figure 3: Partitioning processes

2. *Choose a platform*: construct a platform with the basic components or use a pre-built platform specialized for the type of application to be executed. Figure 4 shows a platform with 2 processors, 2 shared busses, one memory and one transducer (TX in the figure).

3. *Process mapping*: assign a computation element in the platform for each process to be executed. For instance, the main process should be mapped into a programmable processor, and the specialized functions such as filters and mathematical units should be mapped into hardware IP elements (assuming that they will be synthesized into hardware). This is illustrated in Figure 5

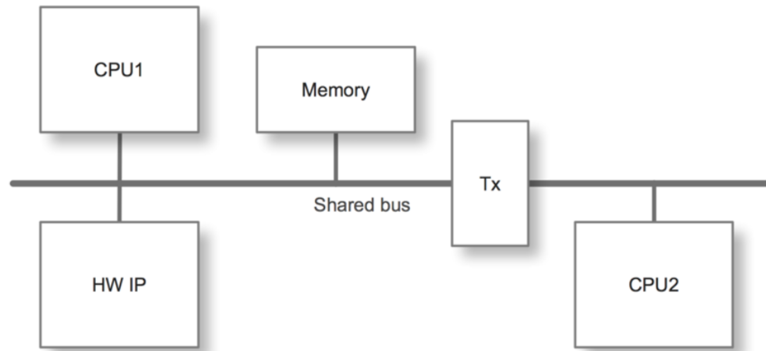

Figure 4: Platform definition

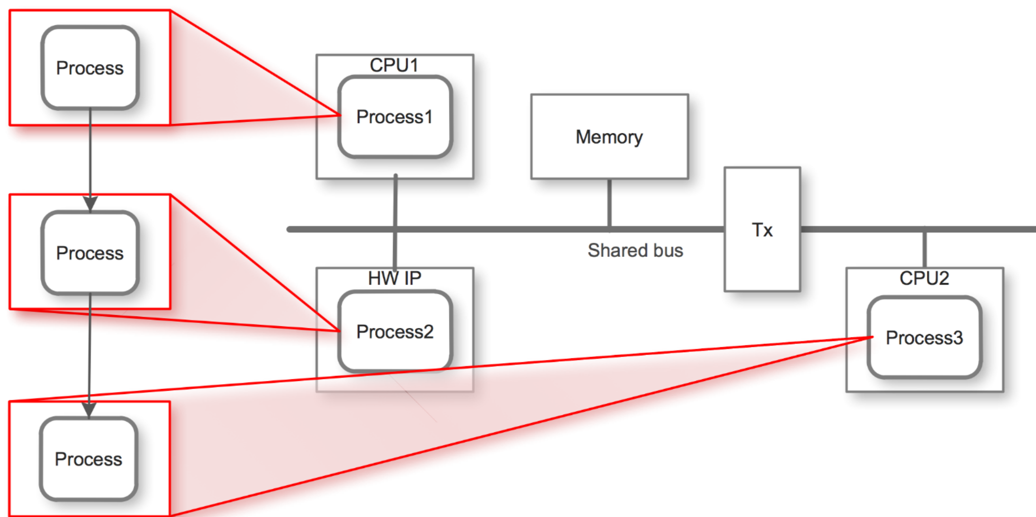

Figure 5: Process mapping

4. *Channel creation and routing*: For each pair of communication processes, a communication channel must be created, an interface must be assigned and a route must be selected. The interface is also called ”process port”, which refers to the specific port which the process will use to communicate with the other process. The process is shown in Figure 6. The last step is route selection, which is the path the data should go to reach its receiving process. All routes may include transducers and one or more busses. In the example shown in Figure 6, channel *Ch1* can go through one shared bus and channel *Ch2* through one transducer and two shared busses.

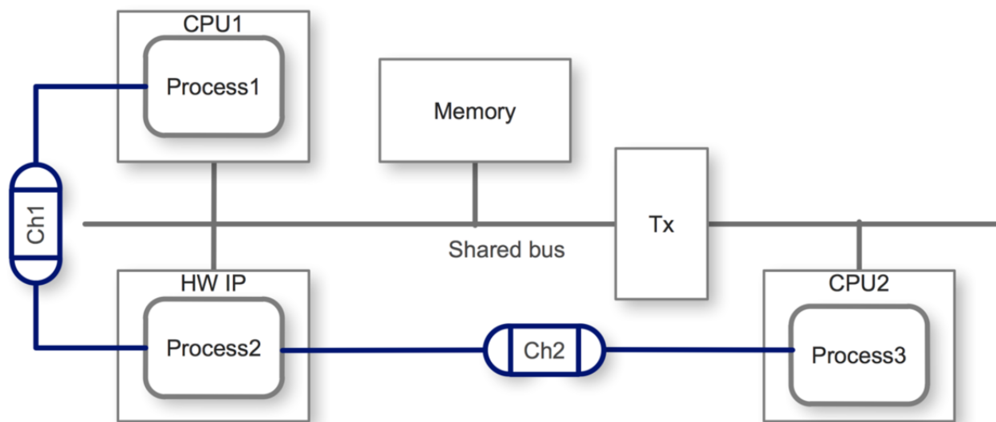

Figure 6: Channel mapping

5 Embedded Systems Environment Data Structure

Our system definition will be stored in a special data structure named *ESE Data Structure (ESEDS)* or *eds* file. This file will contain all information about the application (not the application itself) and all the details about the platform. The reason of having a well-defined and strict (but complete) system data structure is to achieve these goals:

1. *Automatic TLM Generation*: The eds file will contain sufficient information for the generation algorithms to produce an executable SystemC TLM.

2. *Platform transformations*: The user will be capable of transforming the platform in different ways in order to do architecture exploration. The transformations include: adding and deleting objects (processors, communication elements, channels), modifying routes for channels, moving processes from one procesor to another.

3. *PCAM synthesis*: Using the system features defined by the user, our Backend synthesis tool can generate a pin-cycle accurate model of the system.

5.1 ESEDS Formalism

We store ESEDS files in XML format. XML [1] is a markup language that is designed to describe information, it uses plain text and uses user-defined tags to interpret information. XML files are composed of *elements*, which are the building blocks. Each element may have *attributes* which are extra information on the element. Also, an element may be hierarchical: it may contain one or more sub-elements under it.

The sets of rules which the ESEDS must conform to is described in an XML Schema Definition (XSD) [2]. The schema defines the set of elements of a ESEDS, the set of valid attributes of each element, the set of child elements under a hierarchical element, the data types for elements and attributes, the default and fixed values for elements and attributes and the number of child elements.

The schema definition defines which *types* of elements can a ESEDS have, not the number, since that is defined in the XML file. Since it also defines the type of subelements each element has, the best way to illustrate this is with a tree diagram, in Figure 7.

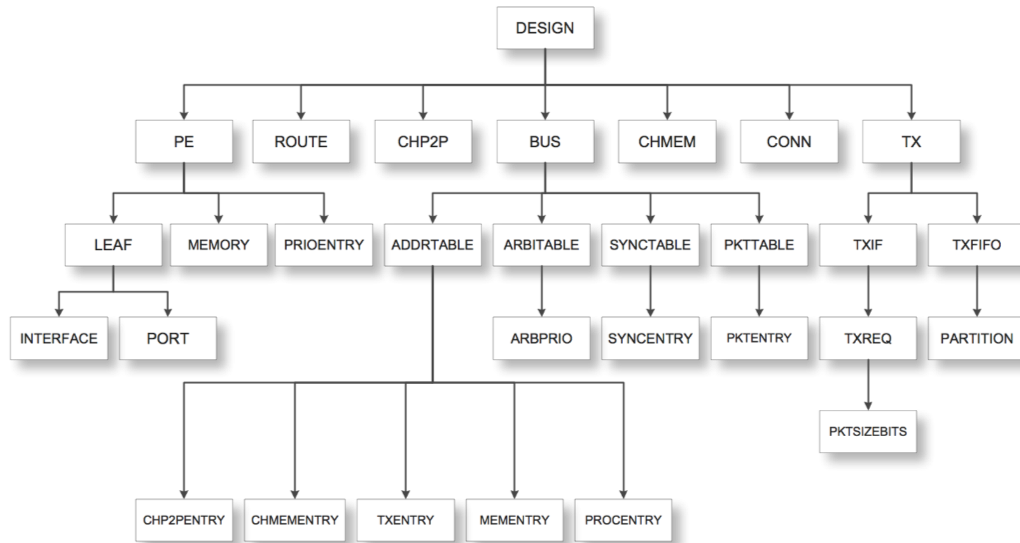

Figure 7: ESEDS element hierarchy

All objects contain a set of *attributes* and may contain a set of children objects. The only object in the ESEDS files is the *DESIGN* element, which contains all other objects.

5.1.1 Design

The *DESIGN* element is the root element of all ESEDS documents. It can be said that it is a set of processing elements (PEs), memories and hardware IPs, and the communication elements between them (transducers). It also contains connections and channel information.

Formally, the objects of a *DESIGN* d are: $< P, B, T, C, R, Ch_{p2p}, Ch_{mem} >, Ch_{fifo}$, where

- P is the set of Processing Elements
- B is the set of Busses
- T is the set of Transducers
- C is the set of Connections
- R is the set of Routes
- Ch_{p2p} is the set of Process-to-process channels
- Ch_{mem} is the set of Memory channels
- Ch_{fifo} is the set of Fifo channels

Its attributes are set by $(n_d, b_d, c_d, l_d, s_d, p_d)$ where n_d is the name of the design, b_d is the FPGA board model, c_d, l_d, s_d and p_d are the compilation, linking, simulation and post-simulation options, respectively.

Its whole tree is shown in Figure 7. A simple example with one PE, one bus, one transducer and the connections between them is shown below:

5.1.2 Processing Element (PE)

As shown in listing 1, a PE is a child of the *DESIGN* element and is a processor. Other categories are hardware IP or memory. Its children are defined formally by $< L, I, M >$:

- L: represents the set of processes running inside the PE. It states the C files plus header files that compose that process.
- I: represents the ports the process uses to communicate with other processes, via communication channels.
- M: represents the set of memories

Given a PE x, the set of attributes is $(n_x, t_x, s_x, i_x, d_x, db_x)$ where n_x is the name of the PE, t_x is the type, s_x is the synthesis option, i_x and d_x are the instruction and data cache sizes, and db_x is the debug option. A sample PE XML code is shown below: In this example, the *PE0* has one process named *PE0_P0* which is defined by the source files *p1.c* and *fixed.h*. It has a process port (or interface) defined for a send function, which can be used by any selected communication channel.

Listing 1: Sample EDS file

```
1  <!-- ESE API Version:0.1.0b -->
2  <DESIGN name = "bare_ese" board = "VIRTEX2">
3    <PE name = "PE0" category = "PROCESSOR" type = "MICROBLAZE" ismem = "0">
4    </PE>
5    <TX name = "CE0">
6      <TXIF name = "CE0_PORT_0" bus = "Bus0" arbpolicy = "FCFS">
7      </TXIF>
8      <TXFIFO name = "CE0_fifo" size = "0" partstyle = "SPLIT">
9      </TXFIFO>
10   </TX>
11   <BUS name = "Bus0" category = "BUS" type = "OPB">
12     <SYNCTABLE>
13     </SYNCTABLE>
14     <ARBITABLE policy = "FCFS">
15       <ARBPRIO pename = "PE0" prionumber = "0"/>
16     </ARBITABLE>
17     <ADDRTABLE>
18       <TXENTRY name = "CE0 CE0_PORT_0"/>
19     </ADDRTABLE>
20     <PKTTABLE>
21     </PKTTABLE>
22   </BUS>
23   <CONN pe = "CE0" bus = "Bus0" peRole = "S" port = "CE0_PORT_0"/>
24   <CONN pe = "PE0" bus = "Bus0" peRole = "M" port = "MICROBLAZE_PORT_0"/>
25 </DESIGN>
```

5.1.3 Bus

A bus has the children represented by $< AD, AB, SY, PK >$:

1. *AD*: one address table, contains all the high and low addresses for each communication channel, memory, processor and transducer connected to the bus.

2. *AB*: one arbitration table, contains the arbitration table to be used by the bus arbiter; it shows the priority of each process in the bus.

3. *SY*: one synchronization table, contains synchronization entries for each pair of communicating processes. Details information such as synchronization type (interrupt or polling), which process is the initiator or resetter, where is the flag located (resetter or initiator) and the sensitivity (edge or level).

4. *PK*: one packet table, details the packetization options for all channels.

A bus x has the following attributes: (n_x, t_x, p_x) where n_x is the name of the bus, t_x is the type of the bus and p_x is the name of the PE which is parked (if *bus parking* is enabled in this bus).

Listing 2: PE element

```
1 <PE name = "PE0" category = "PROCESSOR" type = "MICROBLAZE" ismem = "0">
2   <LEAF name = "PE0_P0" cfiles = " p1.c" hfiles = " fixed.h">
3     <INTERFACE name = "int1" type = "Send" send = "send_channel1"/>
4   </LEAF>
5 </PE>
```

Listing 3: BUS element

```
1 <BUS name = "Bus0" category = "BUS" type = "OPB" xpos = "10" ypos = "230">
2   <SYNCTABLE>
3     <SYNCENTRY syncid = "0" chname = "ch1" initiator = "CE0_PORT_0"
4       resettor = "mb0" type = "INTERRUPT" flagLocation = "mb0"
5       sensitivity = "None" flagSetMethod = "SIGNAL"/>
6   </SYNCTABLE>
7   <ARBITABLE policy = "FCFS">
8     <ARBPRIO pename = "PE0" prionumber = "0"/>
9   </ARBITABLE>
10  <ADDRTABLE>
11    <PROCENTRY name = "mb0" enable = "FALSE"/>
12    <CHP2PENTRY name = "ch1" pair = "mb0 CE0_PORT_0"/>
13    <TXENTRY name = "CE0 CE0_PORT_0" lowaddr = "0x80000000"/>
14  </ADDRTABLE>
15  <PKTTABLE>
16    <PKTENTRY chname = "ch1" transfertype = "None"/>
17  </PKTTABLE>
18 </BUS>
```

In listing 3, the bus has one channel *ch1* going through it, and has one transducer and one PE connected. This can be seen in the address table (line 9-13). The synchronization table in line 2-5 shows that there is just one synchronization entry, for the only channel *ch1* which communicates processes *CE0_PORT_0* and *mb0*. It is important to note that even though the addresses in the address table must be set manually by the user, the synchronization table is created automatically by the ESE Frontend during channel creationg and route selection.

5.1.4 Transducer (TX)

A transducer relays data transfers from a process in a bus to another process connected to another bus. It is needed when the design hass two hardware modules whose bus protocols are different. It synchronizes with each process before accepting or transfering any data. In the case of a transducer with one communication channel, the generated XML code is shown in listing 4.

The transducer contains one or more transducer interfaces and one FIFO.

Listing 4: TX element

```
<TX name = "CE0" chlist = "ch1" xpos = "70" ypos = "370">
  <TXIF name = "CE0_PORT_0" bus = "Bus0" arbpolicy = "FCFS">
    <TXREQ name = "CE0_PORT_0_req_mb0_mb1_SEND" type = "SEND"
        srcprocs = "mb0" destprocs = "mb1" routes = "RT_0_PE0_PE1"
        txrole = "S" storagesize = "4" place = "LOCAL">
      <PKTSIZEBITS lsb = "8" msb = "31"/>
    </TXREQ>
  </TXIF>
  <TXIF name = "CE0_PORT_1" bus = "Bus1" arbpolicy = "FCFS">
    <TXREQ name = "CE0_PORT_1_req_mb0_mb1_RECV" type = "RECV"
        srcprocs = "mb0" destprocs = "mb1" routes = "RT_0_PE0_PE1"
        txrole = "S" storagesize = "4" place = "LOCAL">
      <PKTSIZEBITS lsb = "8" msb = "31"/>
    </TXREQ>
  </TXIF>
  <TXFIFO name = "CE0_fifo" size = "1000" partstyle = "SPLIT">
    <PARTITION channels = "ch1" size = "256"/>
  </TXFIFO>
</TX>
```

We can see in listing 4 that the transducer contains two transducer interfaces *TXIF*, and one transducer FIFO (*TXFIFO*). The transducer has one channel, and its partition in the fifo has a size of 256 bytes, as seen in line 17. The elements inside the transducer interfaces are the transducer requests (*TXREQ*). These elements describe each communicating channel between processes and also links them to the *ROUTE* element that belongs to them.

5.1.5 Communication Channels

There are three types of communication channels: memory channels (*CHMEM*), process-to-process channels (*CHP2P*) and fifo channels (*CHFIFO*). The channels are abstract elements that reflect the communication between either a memory and a process or between two processes. They do not specify themselves the physical path of the data, this information is described in the *ROUTE* element which is assigned to every mapped channel. All types of channels need an *INTERFACE* from the process. This will be the port from which the process will use to transfer data. Formally, the channel x has attributes defined by (s_x, d_x, r_x, i_x) where s_x is the source process or memory, d_x is the destination process or memory, r_x is the route and i_x is the set of assigned interfaces. Channels do not have any children objects.

5.1.6 Route

A route is assigned to a channel, and points to/from either a processing element or transducer to another. It also lists the physical path through busses and transducers. Formally, a route x has the following attributes: (s_x, d_x, b_x, t_x) where s_x is the source processing element/transducer, d_x is the destination processing element/transducer, b_x is a set of busses, and t_x is set of transducers. Routes do not have any children objects.

5.1.7 Connection

This element represents the physical connection between two objects. The attributes in a connection x are (o_x, b_x) where o_x is a processing element or transducer and b_x is a bus.

5.2 ESEDS Rules

The formal description of ESEDS allows the creation of rules, which will serve to check for correctness in the structure of the EDS file. The ESE frontend must enforce these rules strictly, since the synthesis and generation tools depend on it for their correct funcionality. Given a eds model M and a design d with its objects $< P, B, T, C, R, Ch_{p2p}, Ch_{mem} >$:

- There can be only one Design element in each EDS file
- Processing Elements (PE) can only be connected to one bus
- Each route can be assigned to only one channel
- Process-to-process channels must be linked to two processes
- Memory channels must be linked to one process and one memory
- Each route must have one bus and any number of transducers
- For every channel that goes to a processing element, there is a port assinged to it
- For every channel that goes through a bus, it must have an address range set in an address entry in its address table , a synchronization entry in its synchronization table and a packet entry in its packet table
- For every channel that goes through a transducer, it must have at least two transducer request entries in its transducer interface and a partition in its transducer fifo.

6 Platform Manipulations

Having our model in a well-defined data structure, allows us to develop tools that will manipulate the system to fit our needs. Some of these needs include using the EDS file to produce and executable system level model and transforming the platform (an thus the system) to explore different architecture options.

6.1 Basic platform operations

The basic operations on the platform are:

1. Loading EDS files from disk
2. Saving EDS files to disk
3. Exporting EDS designs to disk
4. Adding EDS objects to the platform
5. Deleting EDS objects to the platform

Loading and saving EDS files involves not only the EDS files themselves, but also a set of files attached to each process (C and H files) in a directory hierarchy that represents our design. In Figure 8 we can see how an EDS project is saved.

The export operation gathers all theses files and compresses them into a single .bzip file, preserving this directory hierarchy.

6.2 Moving processes

One possible operation the user might use is to move a process from a PE to another PE. This need might arise when doing design exploration: the process may be slowing down a specific PE because of an heavy load of processes in the Real Time Operating System (RTOS) . This can be seen using the computation estimation tool during the SystemC simulation of the TLM. In this case, the user can move the process to another embedded processor (PE) to lighten the load and speed up execution time of the system.

This process has to take into account several dependencies between ESE objects. The changes needed in the platform are:

1. Moving the process from source PE to target PE
2. Moving all ports that involve that process
3. Preserve the underlying directory/files structure of the PEs and processes.

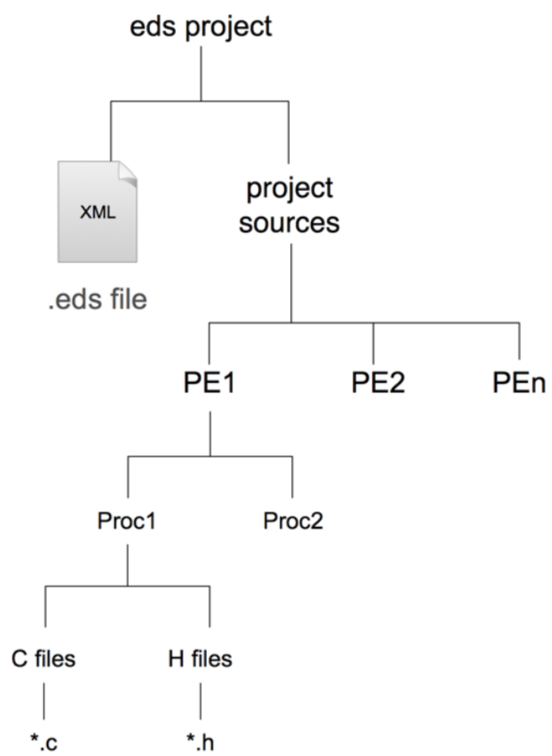

Figure 8: ESE DS project directory structure

6.3 Application plus mapping specification

Mapping refers to the process of assigning either a process to a PE or a physical route to a CHP2P or CHMEM. To assign a process to a PE, the procedure consists of creating a LEAF and assigning C files and header files to it. This was illustrated in Figure 5.

In the case of channel mapping, it refers to the process of assigning a route to the channels. There are two different types of routing:

1. Dynamic Routing: the route is selected at run-time. The packet is directed to the next node (transducer) that is free and is closer to the final destination.

2. Static Routing: the route is selected at compile-time, so the path of the data is already known and cannot change regardless of the circunstances.

The approach we have chosen is static routing, so the user must decide which route the channel will have.

To map a route to a given channel, the user has to select the set of busses and transducers that will be linked to the channel. Several options may exist for this step, since data can travel by several different routes to reach its destination (either a memory or another process). We can see in Figure 9 that for a channel *ch1*, there are two different routing options: either the red path, which will go through bus2, transducer2, bus1, transducer1 and finally bus3, or the green path, which is shorter, since it'll go through bus2, transducer3, and bus3.

6.4 ESE Object configurations

The ESE frontend shall also allow setting configuration options on each element. There are multiple options for each element, we list below the most important:

1. Compilation and linking options for the design and for each process

2. Synchronization type and sensitivity for each channel

3. Synchronization flag location for each pair of communicating processes

4. Packet size for each channel

5. Transducer fifo type: shared fifo or split fifo

6. Partition sizes for split fifo

7. Arbitration policy for busses

8. Process priorities

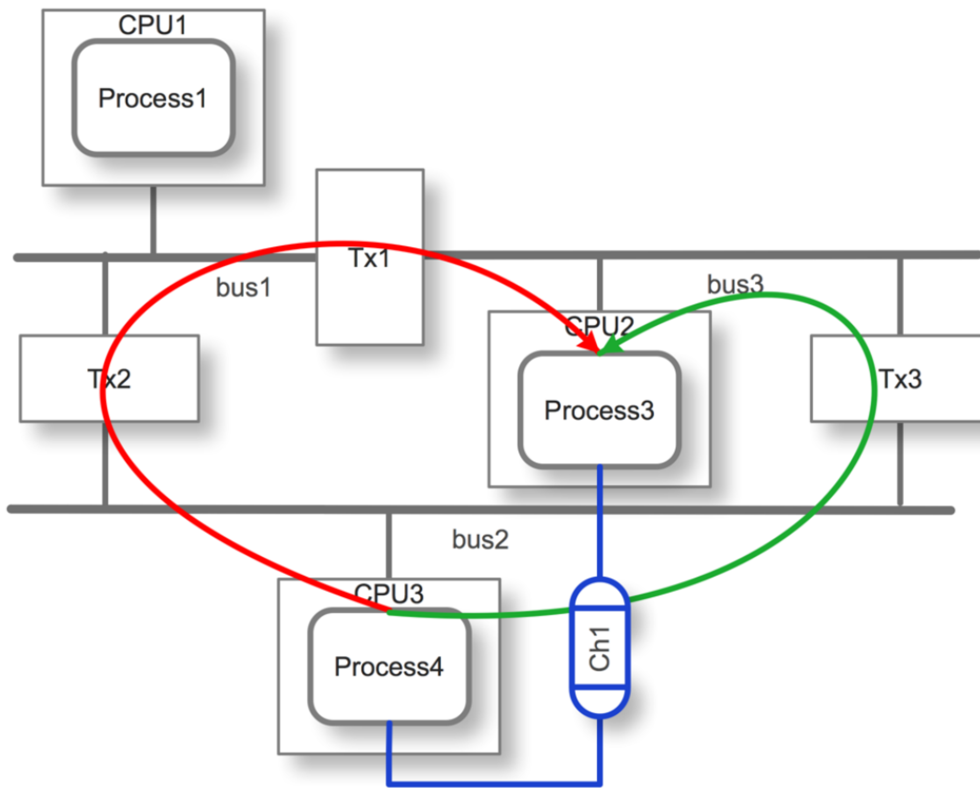

Figure 9: Routing options for a channel

7 Experimental Results

In order to test the performance of the ESEDS and our tools, we chose 3 multimedia applications: a JPEG decoder, a MP3 decoder and a H264 decoder. The test setup for each application is composed of two phases:

1. A simple one-processor platform is created, hosting all processes inside. The data structure is saved, exported before running the functional TLM generator and the TLM estimator tools. Finally, the timed TLM is run and the estimated timing is recoded.

2. Several processors are added into the system, along with one or more shared busses and transducers, this is the platform transformation step.All processes are distributed evenly between all processors. Next, the communication channels are created and routes are assigned to each one of them (remapping step). The TLMs are generated again and timing is recorded.

Every step in each phase has its time recorded, and it does not account for the user's response time, but only for the tool's execution time. All tests were performed on an Pentium 4, 3 GHz, 1 GB RAM machine running Linux kernel 2.6.

The Jpeg application consists of 5 processes, the mp3 decoder has 3 and the h264 decoder consists of a total of 13 different processes.

The transformed platforms (after phase2) of the multimedia applications are shown in Figure 10, Figure 11 and Figure 12. The boxes represent processors and transducers, the rounded boxes represent the processes and the lines present the busses and their connections. The channels and routes are not represented in the figure.

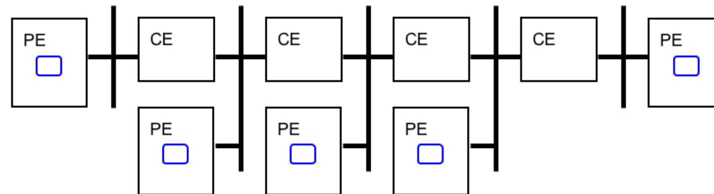

Figure 10: JPEG decoder platform

The experimental results are presented in Table 1. As we can see, the basic operations' (*load*, *save* and *export*) time the same order or magnitude in the simple and parallel platforms (see also Figure 13). The H264 platform is much more complex, and this is reflected in the higher load and save times. This can be seen also in the platform transformations and remapping results (Figure 14). The H264 has significantly more channels, so it's natural to see that the remapping time is much higher.

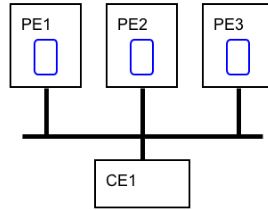

Figure 11: MP3 decoder platform

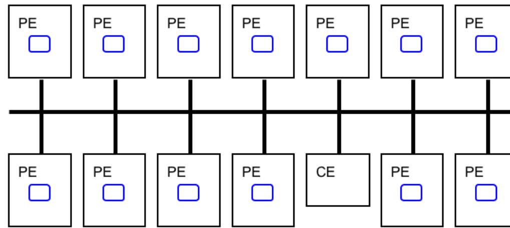

Figure 12: H264 decoder platform

Table 1: Platform operations experimental results

	Platforms					
	JPEG-1	JPEG-2	MP3-1	MP3-2	H264-1	H264-2
XML LOC	16	281	35	219	489	503
XML bytes	477	15396	3168	15806	39151	43123
Load (s)	2.286	3.026	4.561	7.496	14.063	14.656
Save (s)	2.894	3.030	10.131	17.064	35.070	36.451
Export (s)	0.443	1.401	1.156	1.288	4.022	4.862
Platform transformation (s)	-	0.914	-	0.695	-	10.310
Remapping (s)	-	0.528	-	0.326	-	6.584
Func TLM Gen (s)	1.181	3.001	1.062	0.846	1.144	2.053
Timed TLM Gen (s)	22.660	32.5	83	109	168	179
Estimated timing (cycles)	$72.9 \cdot 10^6$	$15.8 \cdot 10^6$	$3.51 \cdot 10^9$	$2.66 \cdot 10^9$	$6.94 \cdot 10^{11}$	$1.35 \cdot 10^{11}$

In the case of function generation of TLMs (Figure 15), the time ranges between 1 and 3 seconds, and taking into account the variability of the system's resources, we could say that the generation time is approximately the same for all platforms.

For the timed TLM generation, the differences are bigger because of the annotation time. This is proportional to the size of the source code, so the bigger the application, the longer it takes (Figure 16). We can see in the last row of Table 1 that the estimated execution time of the multimedia applications (in cycles) indeed goes down after the platform is transformed and remapped to a parallel one. In Figure 17 we can see the percentage of speedup after the transformations.

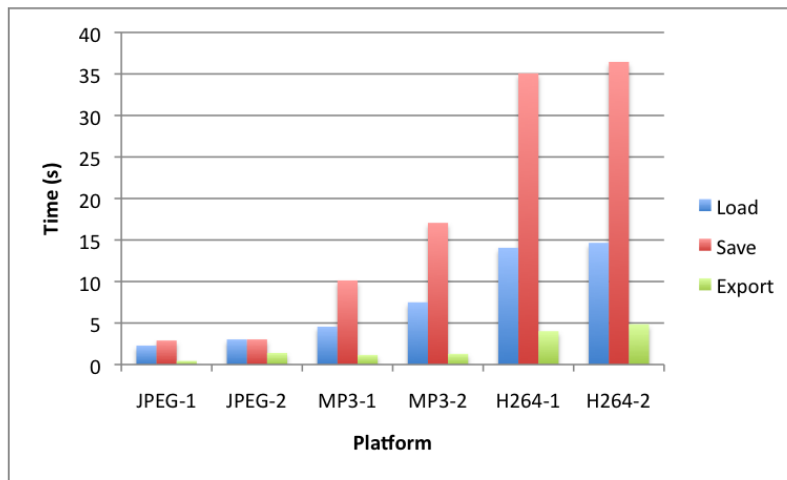

Figure 13: Basic ESE Operations

8 Conclusions

In this report, we presented the description of the Embedded Systems Environment Data Structure (ESEDS), its formalism and rules. As a summary, we can say that:

- ESEDS provides a complete description of the system: application and platform, enough to manipulate the platform to transform it and explore new platform configurations.
- The basic operations on the platform, mapping and platform transformations can be done in a small amount of time, allowing the designer to focus on design space exploration.
- The well structured data structure allows other tools to be developed to take the system specification as input and produce either a functional TLM, timed TLM or cycle-accurate model.

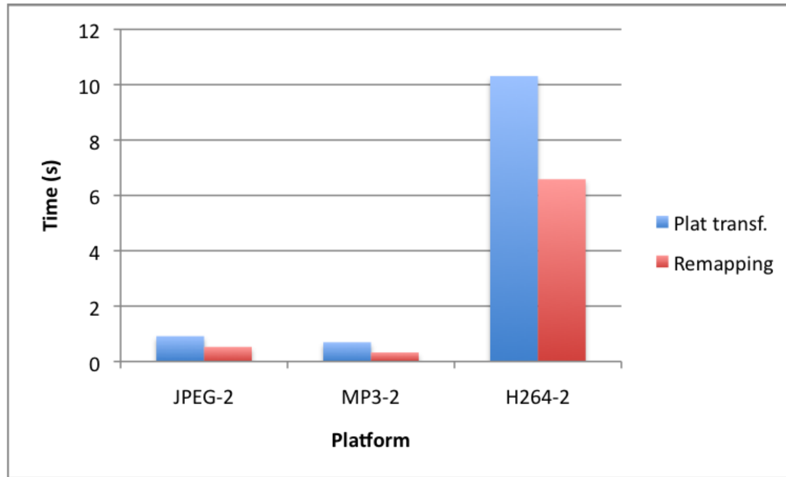

Figure 14: Platform transformations and mapping

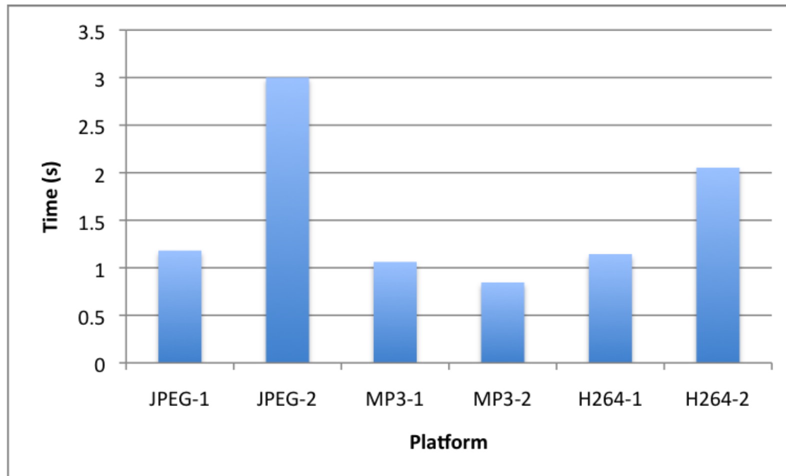

Figure 15: Functional TLM Generation

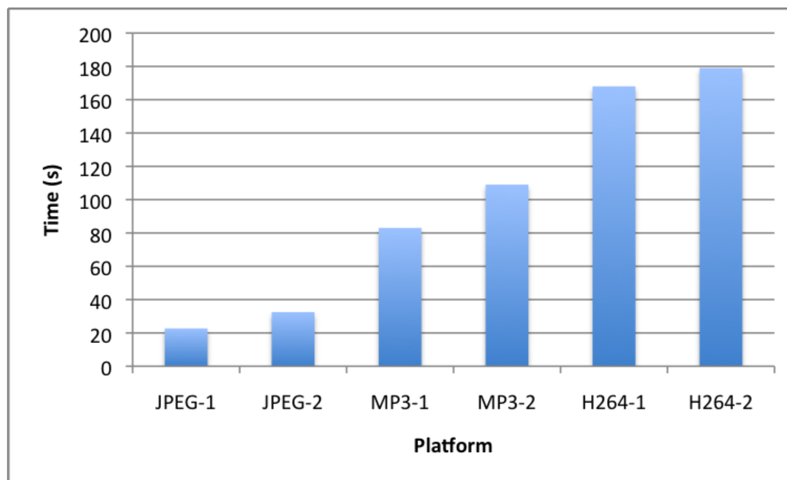

Figure 16: Timed TLM Generation

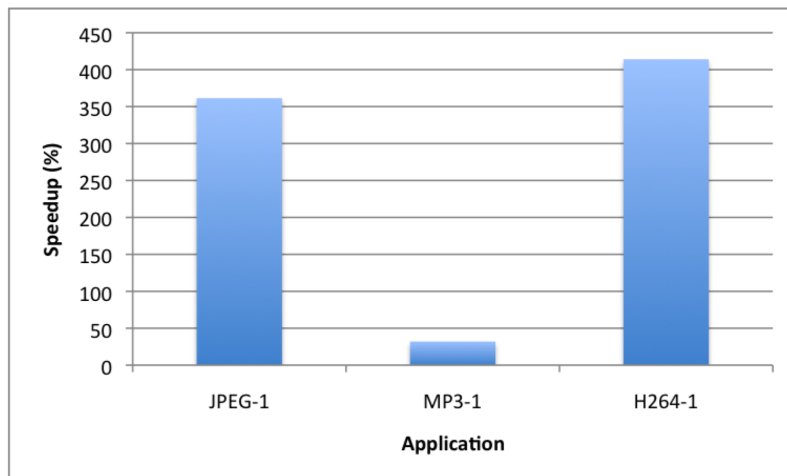

Figure 17: Execution speedup after platform transformations

9 Acknowledgments

This work could not have been done if not for the work of Junyu Peng who laid the foundations of ESE Data Structure. Yonghyun Hwang, who built the first part of the ESE API tool and also helped define the ESEDS, deserves credit for his contributions to this work.

References

[1] Xml. http://www.w3.org/XML.

[2] Xml schema. http://www.w3.org/XML/Schema.

[3] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Kyushu University, Japan, 1996.

[4] Luka Cai and Daniel Gajski. Transaction level modeling: an overview. In ACM Press, editor, *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, pages 19–24, New York, NY, 2003.

[5] L. Carloni, F.D. Bernardinis, C. Pinello, and A. Sangiovanni-Vicentelli. *Embedded Systems Handbook*, chapter Platform-based design for embedded systems. CRC Press, 2005.

[6] CECS. Embedded systems environment. http://www.cecs.uci.edu/ ese, 2008.

[7] E.M. Dashofy, A Van der Hoek, and R.N. Taylor. A highly-extensible, xml-based architecture description language. In *Proceedings of the working IEEE/IFIP Conference on Software Architecture*, pages 103–112, 2001.

[8] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the Design, Automation and Test in Europe conference*, March 2008.

[9] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vicentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12):1523–1543, December 2000.

[10] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3:1, 1998.

[11] Juncao Li, N.T. Pilkington, Fei Xie, and Qiang Liu. Embedded architecture description language. *Computer Software and Applications, 2008 COMPASA '08, 32nd Annual IEEE International*, pages 32–43, July 2008.

[12] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. In *IEEE Proceedings on Computers and Digital Techniques (CDT), Special Issue on Embedded Microelectronic Systems: Status and Trends*, volume 152, pages 285–297, May 2005.

[13] Alberto Sangiovanni-Vicentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[14] Frank Vahid and Tony Givargis. Platform tuning for embedded systems design. *IEEE Computer*, 34(3):112–114, March 2001.

[15] Lucky Lo Chi Yu Lo and Samar Abdi. Automatic systemc tlm generation for custom communication platforms. In *Proceedings of 25th IEEE International Conference on Computer Design*, October 2007.