Moazzam Fareed Niazi | Tiberiu Seceleanu | Hannu Tenhunen

An Automated Control Code Generation Approach for the SegBus Platform

Turku Centre *for* Computer Science

TUCS Technical Report
No 981, August 2010

An Automated Control Code Generation Approach for the SegBus Platform

Moazzam Fareed Niazi
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
moazzam.niazi@utu.fi

Tiberiu Seceleanu
ABB Corporate Research, and
Mälardalen University
Västerås, Sweden
tiberiu.seceleanu@se.abb.com

Hannu Tenhunen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
hannu.tenhunen@utu.fi

TUCS Technical Report
No 981, August 2010

Abstract

We present here a model-driven approach for the generation of low-level control code for the arbiters, to support application implementation and scheduled execution on a multi-core segmented bus platform, SegBus. The approach considers Model-Driven Architecture as a key to model the application at two different abstraction levels, namely as Packet-Synchronous Dataflow and Platform Specific Model, using the SegBus platform's Domain Specific Language. Both models are transformed into Extensible Markup Language schemes, and then utilized by an emulator program to generate the "application-dependent" VHDL code, the so-called "snippets". The obtained code is inserted in a specific section of the platform arbiters. We present an example of a simplified stereo MP3 decoder where the methodology is employed to generate the control code of arbiters.

Keywords: Control Code, Application schedule, Arbitration, Domain Specific Language, UML, SegBus, Model Transformation

TUCS Laboratory
Distributed Systems Design

1 Introduction

The decreasing technological figures cause modern day designers to move towards on-chip multiprocessing technologies. New architectures are brought into context in order to utilize the tremendous advances of fabrication technology. Distributed on-chip architectures or multiprocessor system-on-chip (MPSoC) paradigm gains increasing support from system designers. MPSoC is seen as one of the foremost means through which performance gain are still to be sustained even after Moore's law may become decrepit [1]. The most common MPSoC platforms are *network-on-chip* (NoC) [2], and *segmented bus* platforms [3][4].

As the complexity of the application requirements is increasing with time, the designers are facing difficulty while designing applications targeting MPSoC. However, it has also been a challenge to fully benefit from the features of MPSoC platforms. One of the reasons behind the difficulties in MPSoC development is the deficiency of design methodologies [1]. The current design methodologies doesn't provide full automation in every level of the development process, and sometimes, the communication characteristics of the platforms and the employed devices also do not match from system requirements. In order to offer an optimum match, platform specific characteristics must be taken into consideration for each application.

Model-to-text (M2T) transformation [5] plays a key role in *Model-driven architecture* (MDA) based development [6]. The outcomes produced by M2T usually are textual artifacts from the provided graphical models. These textual artifacts could be in the form of XML schema or source code of any desired high-level programming language like C++, Java, etc. The XML schema provides means for defining the content, structure and semantics of XML documents.

The approach we deliver in this report is based on establishing a design methodology for MPSoC, in the context of the *SegBus* platform [4]. In our previous work [7][8], we have already introduced a *Domain Specific Language* (DSL) and an *emulator* program for modeling and emulating applications, targeting the *SegBus* platform. The previous work was limited only to modeling and emulation using the *Platform Specific Model* (PSM)-level models. We deliver here methods to update the DSL and emulator to make them capable for the modeling of application at *Packet-Synchronous Data Flow* (PSDF)-level, and introduction of automated methods within emulator program to generate transaction-level control code in the form of VHDL *snippets*, in order to successfully implement a given application on the distributed platform at hand. We introduce procedures to transform PSDF and PSM models into XML schemes with the help of modeling tool [9]. The generated XML of the PSDF and PSM models are then used by the *emulator* [8] program to assess the performance aspects. If we find the performance aspects up to an optimum level, the current research work addresses issues how we generate the transaction-level control code from the emulator program in an automated way.

The generation of control code and their realization is especially necessary as the platform doesn't require (or benefit) from an operating system solution. Seceleanu et. al. [10] provided definition of the *SegBus*'s arbiters' control structures at two different (segment and central) levels. The definition comes in the form of VHDL code snippets that provide the transfer schedule, such that arbiters at segment and central levels organize the execution following the application specification. The delivered approach to define control structures is based on PSDF, too, but it is built manually. In this report, we continue our efforts towards an automated design framework.

Related Work. In recent years, MDA has been utilized in different design areas to provide automation up to some extent. Vidmantas et al. [11] introduced MDA methods where the designer can model application as PIM model using UML together with SysML plugin. They introduced techniques to transform PIM into PSM model, which is later transformed into source code specifically for one operating system (OS). The authors have considered more than one OS where the modeled application can be run, unlike our case where there is no consideration of OS is required.

Koudri et al. [12] presented design flow for System-on-Chip / System-on-Programmable chip design, based on the use of UML and dedicated profiles. They supported the use of the Model-Driven Development for the hardware and software co-design with an example of *Cognitive Radio Application*, implemented on FPGA. The modeling tool they used generated thousands of lines of code for the modeled example application but further improvements needs to be done, particularly in the Model of Computations support.

2 Background

2.1 Segmented Bus Architecture

A segmented bus is a "collection" of individual buses (segments), interconnected with the use of FIFO like structures. Each segment acts as a normal bus between modules that are connected to it and operates in parallel with other segments. Neighboring segments can be dynamically connected to each other to establish a connection between modules located in different segments. Due to the segmentation of the bus lines, and their relative isolation, parallel transactions can take place, thus increasing the performance. A high level block diagram of the segmented bus system which we consider in the following sections is illustrated in Figure 1.

The *SegBus* communication platform is built of components that provide the necessary separation of segments - *Border units* (***BU***), arbitration units - the *Central Arbiter* (***CA***) and local, *Segment Arbiters* (***SA***). The application then is realized with the support of (library available) *Functional Units* (***FU***).

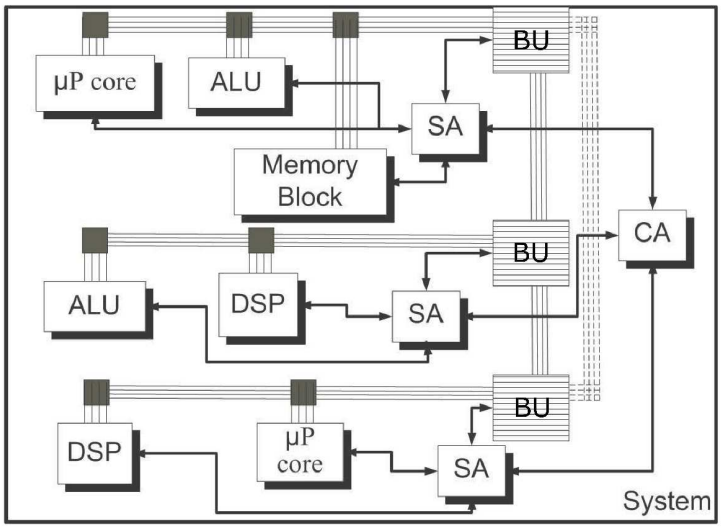

Figure 1: Segmented bus structure.

The *SegBus* platform has a single ***CA*** unit and several ***SA***s, one for each segment. The ***SA*** of each bus segment decides which device (***FU***), within the segment, will get access to the bus in the following transfer burst.

Platform communication. Within a segment, data transfers follow a "traditional" package based bus protocol, with ***SA***s arbitrating the access to local resources. The inter-segment communication, is also a package based, circuit switched approach, with the ***CA*** having the central role. The interface components between adjacent segments, the ***BU***s, are basically FIFO elements with some additional logic, controlled by the ***CA*** and the neighboring ***SA***s. A brief description of the communication is given as follows.

Whenever one ***SA*** recognizes that a request for data transfer targets a module outside its own segment, it forwards the request to the ***CA***. The later identifies the target segment address and decides which segments need to be dynamically connected in order to establish a link between the initiating and targeted devices. When this connection is ready, the initiating device is granted the bus access, and it starts filling the buffer of the appropriate bridge with the package data. Following a signaling protocol, the data is taken into account by the corresponding next segment ***SA***, which forwards it further, towards the destination. At this point, the ***SA*** of the targeted segment routes the package to the own segment lines, from where it is collected by the targeted device.

The arbitration at ***CA*** level implements the application data flow, with respect to these transfers. Hence, one has to implement accurate control procedures for inter-segment transfers, as possible conflicting requests must be appropriately satisfied, in order to reach performance requirements and to correctly implement applications.

2.2 DSL for the SegBus Platform

The *Domain Specific Language* (DSL) for the *SegBus* platform is the specification language that is used to model the *SegBus* platform at higher-level of abstraction, based on stereotypes stored in the *SegBus* UML profile [7]. The DSL provides ability to model platform elements in the form of high-level graphical constructs and provide methods to map partitioned application components on particular segment in a fast and correct manner.

The DSL comprises of a number of structural constraints related to the platform, written in *Object Constraint Language* (OCL) [13], to implement the correct component approach to platform design. These constraints are used to validate our models. Upon breach of any constraint requirement during the design process, the tool provides appropriate error message, so that the designer can take proper action to make the model correct according to platform requirements.

The relationship between all platform elements are defined using *Customization* classes. The customization classes comprise of tags that store user-defined DSL customization rules. The customization rules are parsed and interpreted by the *DSL Customization Engine* (provided by the tool) to assist the validation process.

Before the current work, the DSL was only capable of modeling application at PSM-level. Here, we add capabilities to model application at PSDF level, too. We introduce three new stereotypes, that is, *InitialNode*, *ProcessNode* and *FinalNode*, in the UML profile of DSL. The profile defines the main structural elements of the platform. The new stereotyped classes related to PSDF are generalization of the metaclass *UML Standard Profile::UML2MetaModel::Classes::Kernel::Class*. We also introduced their related customization classes and set tags with suitable values. We skip here further details about tag values intentionally because of the space limitation.

Once we model the application components as PSDF, model the platform and map the application components on to the platform correctly, we apply validation process to get the correct *Platform Specific Model* (PSM) of the application. If there exists some errors in the model, we get error message(s) and associated model element become highlighted.

Finally, the PSDF and PSM model can be transformed into XML schema for further analysis of the desired platform configuration. We employ the generated XML schemas for emulating the performance aspects of the configured system, as described in the next section.

2.3 SegBus Emulator

The *SegBus Emulator* enables us to evaluate the performance aspects of any given application running on a specific platform configuration, defined during modeling [8]. The emulator supports the analysis of various *SegBus* instances that may

answer, better or worse, to specific application requirements. It helps to decide at early stages of design process which platform configuration will be most suitable for any given application before moving towards lower abstraction levels. The code generation engine, supplied by the *MagicDraw UML* [9] tool transforms PSDF and PSM of the system into XML schemas. The generated XML schemas are then employed by the emulator program to estimate the utilization of platform elements with respect to data transfers and total execution time. After the analysis of the returned results, the designer is able to make decision at this stage whether the emulated configuration will be best/optimal or not, for the target application, and can change it before moving towards lower levels of the design process. After getting the desired platform configuration for a given application, the next step is to generate the execution schedule in the form of VHDL snippets, to be later used by the arbiters.

3 Design Methodology

In this section, we discuss our approach of modeling, transforming and generating the arbiters' control code using DSL and *SegBus* emulator. We employ the *MagicDraw UML* [9] tool for graphically modeling the application at PSDF (discussed in section 3.1) and PSM level, and transforming it into XML schemas. Figure 2 illustrates the design methodology employing DSL and emulator. We demonstrate our approach with the help of a (simplified) stereo MP3 decoder [14] application.

3.1 The Packet SDF

The specification of the application itself starts with a *Packet SDF* (PSDF) model. PSDF is a customized version of Synchronous Data Flow diagrams [15]. The approach is intended to facilitate the mapping of the application to the architecture due to the similarity between the operational semantics of the PSDF and that of the *SegBus* architecture, thus allowing us to cope in a more detailed manner with the communication characteristics of our platform.

A PSDF comprises mainly two elements: *processes* and *data flows*; data is, however, organized in data items, which are later transformed into packets according to package size during execution. Processes transform input data packets into output ones, whereas packet flows carry data from one process to another. A *transaction* represents the sending of one data packet by one source process to another, target process, or towards the system output. A *packet flow* is a tuple of four values, P_t, D, T and C. The P_t value represents the target process for the given transactions; the D value represents the number of data items emitted by the same source, towards the same destination; the T value is a relative ordering number among the (package) flows in one given system; and the C value represents the number of clock ticks a process consumed before sending one package. Thus,

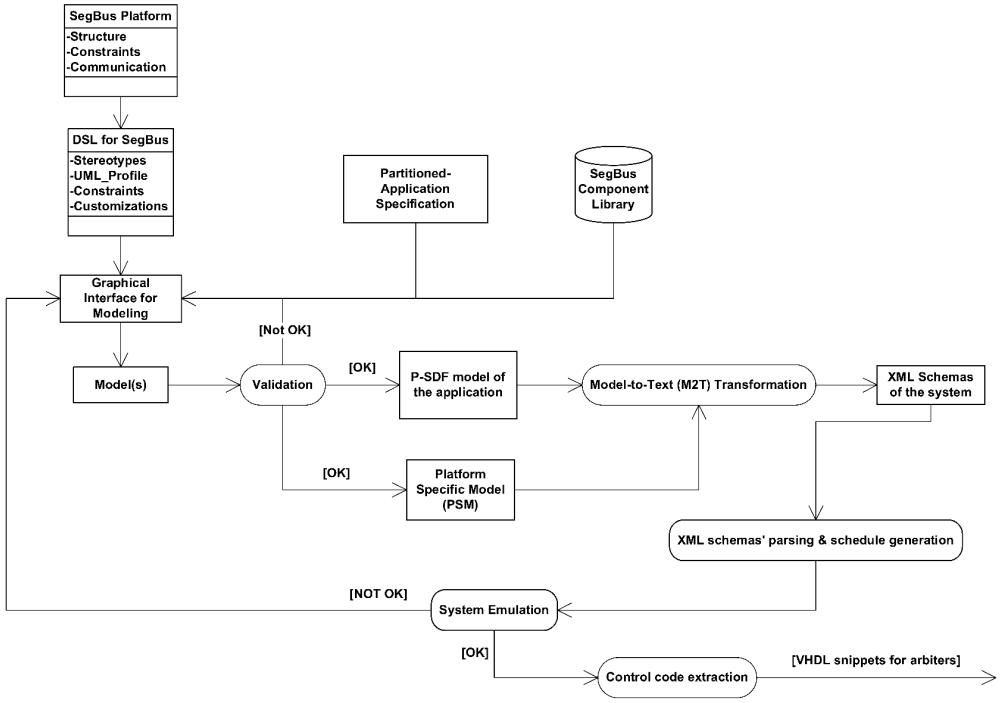

Figure 2: Design process of the SegBus platform using DSL and emulation.

a flow is understood as the number of data items (later transformed into packets) issued by the same process, targeting the same destination, having the same ordering number and same clock ticks require to process one individual package.

If s is the package size (number of data items in a package) in the platform configuration, then the *Packet SDF (PSDF)* of a certain system is a sequence of packet flows, $< (P_{t_x}, \frac{D_1}{s}, T_1, C_1), \ldots, (P_{t_x}, \frac{D_n}{s}, T_n, C_n) >$, where $\forall i, j, x \in \{1, \ldots, n\} \cdot \frac{D_i}{s} \neq \frac{D_j}{s}$ and $T_1 \leq T_2 \leq \ldots \leq T_n$.

The non-strictness of the relation between T values of the above definition models the possibility of several flows to coexist at moments in the execution of the system. In the case of the *SegBus* platform, this most often will describe *local* flows, that is flows where the source and the destination are situated in the same segment. However, considering a segment number larger than 3, *global* flows, where the source and the destination are in different segments, are also possible to be characterized by the same ordering number. In this case, it means that the ***CA***, if possible, allows a simultaneous execution of transactions from all the "same number" global flows.

3.2 Application Modeling

The specification starts with the context diagram of the application, where the interactions between the application (depicted as a process) and the external envi-

ronment are modeled in terms of input/output data-flows. In subsequent steps, the top-level process is decomposed hierarchically into less complex processes and the corresponding data-flows between these processes.

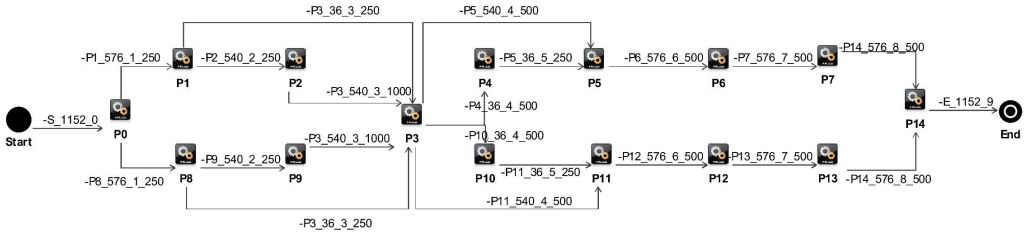

Figure 3: PSDF model of the example application employing DSL.

The decomposition process is based on designer's experience and ends when the granularity level of the identified processes maps to existent *SegBus* library elements or devices that can be developed by the design team. We employed *SegBus* DSL to represent the PSDF . The PSDF model of the example application is given in Figure 3. In brief, process *P0* represents frame decoding, *P1/P8* - scaling on the left/right channel, *P2/P9* - dequantizing left/right channel, etc.

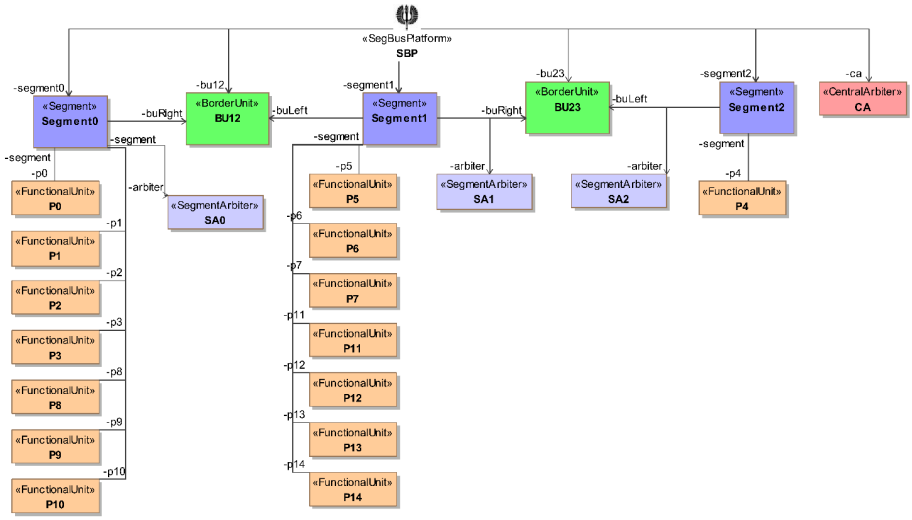

Figure 4: PSM model of the example application in 3 segments, linear topology configuration.

The PSDF model serves as the *Platform Independent Model* (PIM) of the application. We need a further model, that is, the PSM model, to successfully map the application processes on particular segments. To demonstrate our approach, we consider *three* segments platform configuration and map the application processes using the design methods described in [7] and validate both models. Figure 4 depicts the PSM model of the example application. Later on, we transform the

PSDF and PSM models of the application into XML schemas using M2T transformation supplied by the tool. The XML schema contains information about platform elements, application processes in the form of ***FU*** and their relative placement on different segments. The XML consists of a *schema* element and a number of sub-elements, in the form of *complexType* and *element* types.

Each complex type represents a platform element (***CA***, ***SA***, etc.) or application component (P0, P1, etc.). The *name* attribute of each complex type shows the name of the element. Furthermore, each complex type may contain sub-elements. Following, we show an XML snippet of the PSDF model after transformation, consisting of process *P0*, *P1* and their relative transfers to other processes.

```
<xs:complexType name="P0">
     <xs:sequence>
         <xs:element name="P1_576_1_250" type="P1"/>
       <xs:element name="P8_576_1_250" type="P8"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="P1">
     <xs:sequence>
         <xs:element name="P2_540_2_250" type="P2"/>
       <xs:element name="P3_36_3_250" type="P3"/>
   </xs:sequence>
</xs:complexType>
```

Below is the piece of XML snippet of PSM model after transformation, representing the *SegBus* platform instance (*SBP* with three segments as child-elements) and "*Segment 1*" element with its child-elements.

```
<xs:complexType name="SBP">
   <xs:all>
      <xs:element name="segment0" type="Segment0"/>
      <xs:element name="segment1" type="Segment1"/>
      <xs:element name="segment2" type="Segment2"/>
      <xs:element name="ca" type="CA"/>
      <xs:element name="bu12" type="BU12"/>
      <xs:element name="bu23" type="BU23"/>
   </xs:all>
</xs:complexType>

<xs:complexType name="Segment1">
   <xs:all>
      <xs:element name="buRight" type="BU23"/>
      <xs:element name="buLeft" type="BU12"/>
      <xs:element name="p5" type="P5"/>
      <xs:element name="p6" type="P6"/>
      <xs:element name="p7" type="P7"/>
      <xs:element name="p11" type="P11"/>
      <xs:element name="p12" type="P12"/>
      <xs:element name="p13" type="P13"/>
      <xs:element name="p14" type="P14"/>
      <xs:element name="arbiter" type="SA1"/>
   </xs:all>
</xs:complexType>
```

The *communication matrix* is the specification of device-to-device transactions between application components. Each entity in the communication matrix

describe how many data items need to be transfered from one device to any other device. The emulator program builds the matrix by extracting transactions between processes in PSDF model. Based on the matrix, the *PlaceTool* application [16] finds the optimal device allocation solution, given the platform specifics (the number of segments).

The emulation and control code generation processes are based on both PSDF and PSM. The PSDF model provides information about interaction between application processes with required data items and other useful parameters, while the PSM model represents the placement of each application process on different segments of the platform. Hence, the emulator program parses XML of both models to be later used for emulation and control code generation. During the parsing process, the emulator extracts following information from the PSDF model:

- Number of application processes.
- Data transfers from each process.
- Ordering of transfers.
- Clock ticks to be consumed by each process while processing one package.

The emulator stores above information in temporary variables and arrays inside the program. For instance, the name attribute from one of the *element* from *P0*, that is, "P1_576_1_250" represents a transfer from process *P0*. The "_" character serves as the separator between the entities. The first entity "*P1*" represents the target process of this transfer; the second entity "576" is the number of data items to be transferred; the third entity "1" is the sequencing order and the last entity "250" is the number of clock ticks a process needs to consumed before sending each package.

Furthermore, the emulator extracts following information from the PSM model and stores in a number of variables and arrays inside the emulator, too:

- Number of segments in the platform.
- Number of border units based on platform geometry.
- Placement of application processes on different segments.
- ..

When the parsing process is finished, the emulator iterates in the previously populated arrays, instantiates the required ***FU***s and pass them necessary information. This necessary information contains number of data items to be transferred, destination processes, relative ordering, clock ticks a process needs to be consumed before sending a package and placement in the specific segment.

The *contructor* method of the ***FU*** class analyzes the passed information to it and instantiates the required number of objects of *masters* and *slaves*, which later run as threads during emulation. Finally, the values in the temporary variables and arrays within the emulator application are later used for extracting the control code of the arbiters. As per emulator functionalities, we can generate the control code from the supplied XML schemas without performing the emulation, but it is always recommeded to emulate the modeled platform configuration before moving towards the later stages of the design process.

Without considering details, the control flow of both ***SA***s and of the ***CA*** is represented in Figure 5.

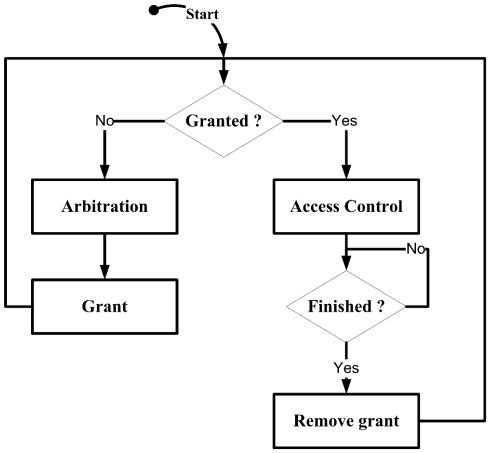

Figure 5: Arbiter control flow.

The ***SA***s and the ***CA*** are VHDL defined modules, with a similar structure. The code implements the operational flow of Figure 5, running with multiple parameters as required by the platform specification. We see the application as a set of correlated transactions that must be ordered in their execution by the arbiters. The specification of the schedule - as supplied by the PSDF representation, is provided by a snippet introduced in the ***SA*** or the ***CA*** codes, representing the projection of the application flow at the respective level and location.

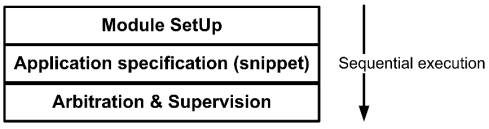

Figure 6: Arbiter code structure.

The intended structure of the arbiters is depicted in Figure 6. The “Module SetUp” and the “Arbitration & Supervision” blocks are concerned with application-independent procedures, such as reading the input signals, selecting the granted

master, counting the number of transactions performed in a granted activity, etc. Our intention here is to generate the middle, “Arbitration specification” block using the automated methods within the emulator program, in such a way that it will bring the application specific requirements for scheduling grant decisions. The resulting snippet will characterize the given application as mapped on a given instance of the platform.

The snippet is part of the actual arbiter VHDL code, and, as such, will be executed. The addressed variables (discussed below) will be read or written by the other arbitration code blocks.

The emulator program reads the package size (36 data items per package in our case), PSDF and PSM models in the form of XML schemes and runs the emulation. Upon completion, the tool returns results of the transactions from each platform element, performed during execution. At this stage, it’s the job of the designer to evaluate the emulation results and modify the design, if needed. Later on, we generate the transaction-level control code (in the form of synthesizable VHDL snippets) of the arbiters to be used in the final implementation. Following, we show the generated control code for the example application after successful emulation.

```
--    VHDL Snippet for "Central Arbiter"
program(0) <= (guard => 0, source => 0,
 dest_seg => 2, togrant => 0, count => 1, enables => 4);
...
program(3) <= (guard => 1, source => 2,
   dest_seg => 1, togrant => 2, count => 1, enables => 0);

--    VHDL Snippet for "Segment 0"
program(0) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
program(1) <= (guard => 0, source => 1, dest => 8,
   dest_seg => 0, togrant => 1, count=16, enables=2);
program(2) <= (guard => 1, source => 2, dest => 2,
   dest_seg => 0, togrant => 2, count=15, enables=3);
...
program(12) <= (guard => 1, source => 8, dest => 11,
   dest_seg => 1, togrant => ToR, count=1, enables=0);

--    VHDL Snippet for "Segment 1"
program(0) <= (guard => 0, source => RFL, dest => 5,
   dest_seg => 1, togrant => RFL, count=15, enables=10);
program(1) <= (guard => 0, source => RFL, dest => 11,
   dest_seg => 1, togrant => RFL, count=15, enables=2);
...
program(9) <= (guard => 1, source => 18, dest => 14,
   dest_seg => 1, togrant => 18, count=16, enables=0);

--    VHDL Snippet for "Segment 2"
program(0) <= (guard => 0, source => RFL, dest => 4,
   dest_seg => 2, togrant => RFL, count=1, enables=1);
program(1) <= (guard => 1, source => 19, dest => 5,
   dest_seg => 1, togrant => ToL, count=1, enables=0);
```

Each line in the above control code is an *execution line* of the respective arbiter. The *program* is a multi-dimensional vector consisting of a number of exe-

cution lines with several further fields. Below is a brief description of each field of the execution line.

• *program(x).* Basically, x can be seen as the *Program Counter*, and *program(x)* represents the *x* line of arbitration code. *x* also provides reference for accessibility from / to other lines of instructions.

• *guard.* When *guard* = 0, the respective line is *enabled*, that is, the arbiter may consider it for selection. When *guard* $>$ 0, the line is disabled, that is, it cannot be considered in the arbitration. The arbiter marks a line as *executed* whenever the respective *count* value reaches 0, by establishing *guard = nrLines*, since *nrLines* is the total number of program lines in the *program* vector, associated with the given arbiter.

• *source.* For ***SA*** case, this field contains the address of the requesting master - the initiator of a transfer request. Devices on the *SegBus* platform (masters, slaves) are identified by an unique number. While for ***CA*** case, this field contains address of the initiating *segment*.

• *dest.* The address of the targeted device - the slave.

• *dest_seg.* The target slave's segment address.

• *toGrant.* This is the instruction for the arbiter to grant the requesting master. At this moment the specification is obsolete, but the field is preserved for future developments.

• *count.* This field identifies the number of packages the master has to send to the specified slave (data items to be sent divided by package size).

• *enables*. Whenever a line is marked *executed*, the ***SA*** will *enable* the execution line specified by this field, by subtracting 1 from its current *guard* value. In order to become enabled, a line with an initial *guard* $>$ 1 will require that several previous operations (execution lines) to have finished. If, for a given line, *enables = nrLines*, then the arbiter does not try to enable any other line, when the current one is marked *executed*.

In addition, we use the notations: ToR/ToL - the destination is the ***BU*** to the right / left of the current ***SA***); RFL - the request comes from left.

The *ArbiterProgram* is a data structure in the emulator application that represents one program line. The source code of this data structure is given below:

```
public class ArbiterProgram {
    public int program;
    public int guard;
    public int source;
    public int dest;
    public int dest_seg;
    public int togrant;
    public int count;
    public int enables;
    public int sequence;
}
```

Each entity in the data structure represents a specific element of the execution line, which we already discussed previously. When the parsing process is done,

the emulator creates:
• The *accCAArray*: a single-dimensional array, where each element in the array represents an execution line of the ***CA***.
• The *accArray*: a 2-dimensional array where each column represents an execution line of a ***SA***, while each row consists of execution lines associated with any particular ***SA***.

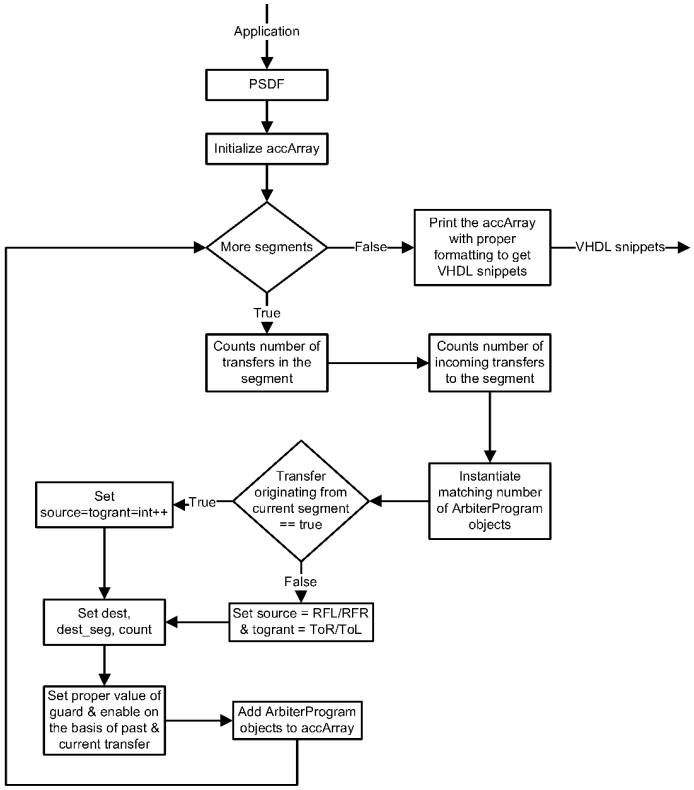

Figure 7: Code extraction process for segments in the platform.

Figure 7 illustrates the general flow of the code generation process for segments after the parsing of the PSDF model has been done and the model resides now inside the emulator's internal variables and arrays. Firstly, the emulator analyzes number of originating and incoming transfers in each segment. On the basis of this information, it creates equal number of *ArbiterProgram* objects. Secondly, it sets the *dest* field with the target process ID and *dest_seg* field with the segment ID where the target process is placed. If the transfer is originated from a master in the current segment, then it sets the *source* value of each object with an integer number in increasing order and *togrant = source*, otherwise the transfer is considered to be coming from a different segment via left/right ***BU***. In this case, the *togrant = ToR/ToL* and *source = RFL/RFR* are set according to the direction of the transfer. The *count* contains number of packages for this transfer (data items divided by the package size). The *program* field contains the order number of the

execution line and the *sequence* field contains the relative order number of the execution line according to PSDF model.

The *guard* and *enables* fields are important to introduce parallelism in the platform. An execution line is executed by the respective ***SA***, when its *guard* signal is zero. The emulator application sets the values of *guard* and *enables* field on the basis of ordering sequence of transfers. If two or more transfers occur at the same ordering sequence, it sets appropriate values to both fields so that parallel transfer can occur. For instance, the PSDF model of the example application in Figure 3 contains two parallel transfers from process *P0* at sequence order 1. As per application requirements, both transfers needs to be completed in parallel before moving towards further transfers. The execution lines associated with these two transfers are given below:

```
program(0) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
program(1) <= (guard => 0, source => 1, dest => 8,
   dest_seg => 0, togrant => 1, count=16, enables=2);
```

A similar approach is taken with respect to the VHDL code to be generated for the ***CA*** operations. The difference is that, instead of considering as source and destinations the actual devices, the ***CA*** code only needs information regarding the initiating segment and the target segment. Hence, the *source* field identifies the requesting segment, and the *dest* field is not necessary.

4 Conclusions

We have introduced MDA-based design methods to generate the transaction-level control code for a distributed platform, the *SegBus*. We have described methods to model application at PSDF and PSM levels by employing *SegBus DSL* and run emulation using *emulator* program to get performance aspects of the modeled configuration. The emulator program has further modified to generate the arbiters' low-level control code, in the form of VHDL *snippets*, which are then to be inserted in a specific block of (segment or central-level) arbiters as an execution schedule for any given application. The approach has now made easy the old manual process of writing such control code for arbitration using presented automated methods.

References

[1] *International Technology Roadmap for Semiconductors.* 2007 Edition.

[2] A. Jantsch, H. Tenhunen. *Networks on Chip.* Kluwer Academic Publishers, 2003.

[3] K. Lahiri, A. Raghunathan, S. Dey. *Design Space Exploration for Optimizing On-Chip Comm. Architectures.* IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, Vol. 23, No. 6, June 2004, pp. 952-961.

[4] T. Seceleanu. *The SegBus Platform - Architecture and Comm. Mechanisms.* Journal of Systems Architecture, Vol. 53, Issue 4, April 2007, pp. 151-169.

[5] *Eclipse Modeling - Model-to-Text (M2T) Transformation.* http://www.eclipse.org/modeling/m2t/

[6] Model-Driven Architecture. http://www.omg.org/mda/

[7] M. F. Niazi, K. Latif, T. Seceleanu, H. Tenhunen. *A DSL for the SegBus Platform.* In proceedings of 22^{nd} IEEE Intl. System-on-Chip Conference (SOCC), 2009, pp. 393-398.

[8] M. F. Niazi, T. Seceleanu, H. Tenhunen. *An Emulation Solution for the SegBus Platform.* In proceedings of 17^{th} IEEE Intl. Conference and Workshops on Engineering of Computer-Based Systems (ECBS), 2010.

[9] MagicDraw UML. http://www.magicdraw.com

[10] T. Seceleanu, I. Crnkovic, C. Seceleanu. *Transaction Level Control for Application Execution on the SegBus Platform.* In proceedings of 33^{th} IEEE Computer Software and Application Conference, 2009, pp. 537-542.

[11] M. Vidmantas, E. Kazanavičius. *Conception of a Multi-Platform System Software and Firmware Development Tool.* Periodical of Information Sciences, Issue 50, 2009, Vilnius University Publishing House, pp. 194-199.

[12] A. Koudri, J. Champeau, D.ulagnier, P. Soulard. *MoPCoM/MARTE Process Applied to a Cognitive Radio System Design and Analysis.* Proceedings of the 5^{th} European Conference on Model Driven Architecture - Foundations and Applications, 2009, pp. 277-288.

[13] OMG. *Object Constraint Language 2.0 Revised Submission, ver. 1.6.* 2003.

[14] C. Park, J. Jang and S. Ha. *Extended Synchronous Dataflow for Efficient DSP System Prototyping.* Journal Design Automation for Embedded Systems, Springer Netherlands, vol. 6, no. 3, 2002, pp. 295-322.

[15] E. A. Lee and D. G. Messerschmitt. *Extended Synchronous Dataflow.* IEEE proceedings, September 1987.

[16] T. Seceleanu, V. Leppänen, O. Nevalainen. *Improving the Performance of Bus Platforms by Means of Segmentation and Optimized Resource Allocation.* The EURASIP Journal on Embedded Systems, Volume 2009 (2009), Article ID 867362, doi:10.1155/2009/867362.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5B, FIN-20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2452-2
ISSN 1239-1891