2021 25th International Conference Electronics | 978-1-6654-4387-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/IEEECONF52705.2021.9467445

DAEDALUS Framework for High-Level Synthesis: Past, Present and Future

Todor Stefanov[1], Hristo Nikolov[1], Lubomir Bogdanov[2, *], Angel Popov[2]
[1]Leiden Institute of Advanced Computer Science, Leiden University,
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
[2]Department of Electronics, Faculty of Electronic Engineering and Technologies,
8 Kliment Ohridski blvd., 1000 Sofia, Bulgaria
lbogdanov@tu-sofia.bg

***Abstract*—The following paper discusses the structure and semantics of an open-source high-level embedded system design framework called DAEDALUS. It consists of multiple tools that help making the transition between the electronic system level (ESL) to register transfer level (RTL) description of streaming data multiprocessor systems. Application, platform, and mapping specifications are thoroughly discussed.**

***Index Terms*—Electronic system level; Multiprocessor embedded systems; Polyhedral process networks; Design space exploration.**

I. Introduction

Developing an embedded system has always been a challenging task due to the fact that hardware and software are being involved in the process and both depend on each other. An improper hardware design would reflect negatively on the software and vice versa. That is why new methods have emerged in the field of embedded systems where both hardware and software are being automatically generated from a high-level system description. Increasing the level of abstraction helps fighting the rapid complexity growth that is connected with that of the design productivity [1]. High levels of abstraction are also used in modelling, simulation and verification that are inseparable part of the entire production process. However, automation may become a challenging task itself when the system abstraction is not well defined, or when system components at specific level are not implemented, or when system design languages do not fit in particular application, etc. To present visually the relationship between different design methodologies and different levels of abstraction a so-called Y-Chart is widely used in the embedded systems development. This chart is proposed by Gajski [2]. It presents modelling of a design, no matter how complicated it is, in three basic ways (hence the three arcs that resemble the letter Y from the Latin alphabet) – behaviour, structure and physical design (Fig. 1). The term "behaviour" is also referred to as functional model or specification. It describes the system as a black box whose inputs are known in advance and its outputs respond in a specific way. As the input stimuli change over time, the outputs respond in a correlated manner. The inner mechanisms that make the outputs change are not described at all. Those mechanisms are shown on the structure axis that is sometimes referred to as a block diagram or a netlist. It must contain separate blocks that are connected to each other in such a way that the whole could perform a certain task. What is missing so far are parameters that describe the components – size, position, ports, connections, etc. In a real world, this would mean the layout of a silicon chip or a printed circuit board. To further expand the Y-Chart, each axis contains different levels of abstraction of the selected design that are shown as circles around the center of the chart. In most cases, up to four levels are used:

– Circuit level
– Logic level
– Processor level
– System levels.

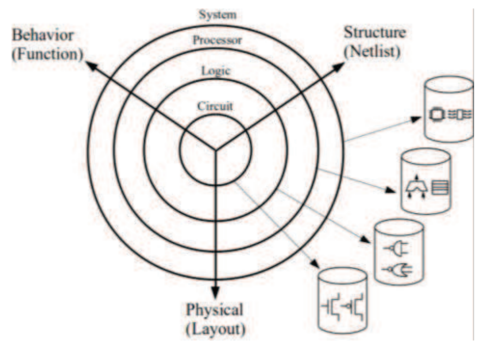


Fig. 1. The Gajski Y-Chart.

Those names are derived from the components that are automatically generated at the end of the respective level. Circuit level uses basic blocks such as transistors, resistors, capacitors, diodes and so on. Combined together they make up a circuit. In digital systems, circuits that implement basic logic functions (such as AND, OR, XOR, NOR, etc) are called gates and they are the main building blocks of the logic level. Complex blocks may group together to form registers, ALUs, multipliers and other functional and memory elements. This level is also referred to as Register Transfer Level, or RTL. The elements from the RTL combine to form processing elements and state machines such as standard processors, memory controllers, arbiters, bridges, and interfaces. The system level uses processors, memories, buses, and other high-level components as main

Manuscript received 30 January, 2021; accepted 11 April, 2021.

building blocks. They are produced as a result of the design at the RTL. The abstraction at this point is so high, that the developers do not need to know any low-level details such as how are the gates connected, or how are the registers grouped in files. The entire thought is focused on what should the system do, given a specific input, and what should the response be in that specific case. This level is also referred to as Electronic System Level, or ESL.

II. The Daedalus Framework

Nowadays many development environments for ESL synthesis exist. Some of them are freeware, others are commercial and require paid license. Industrial giants such as Mentor Graphics, Cadence and Synopsys provide ESL tools. An open-source alternative is the Daedalus design flow (http://daedalus.liacs.nl) that originates from the Leiden Institute for Advanced Computer Science (LIACS) at Leiden University, The Netherlands. The main tools involved in the synthesis are shown in Fig. 2. Daedalus fills the so-called "implementation gap" in system-level design – it bridges the ESL and RTL levels by providing a tool called ESPAM. Other manufacturers either have ESL frameworks, or RTL frameworks, but none have a framework that combines both in a single tool flow. Daedalus is suited for designs that require streaming data processing and are implemented on multi-processor system-on-chip (MPSoC). Certain rules have to be followed, so that the C program is compliant with Daedalus. A list of those rules is given section V. Designs are easily prototyped on an FPGA and verified. SystemC timed simulations are also available, as we will see later in this paper. System design begins with functional description of the system. Currently the standard ANSI C is supported but the front-end could be modified to support any other specification language (SystemC, SpecC, etc). The C language is unchanged, i.e. no modifications to the C syntax are needed. Certain coding style rules have to be respected. The program is a sequential one at this stage (which is easier for the developer). Next, an ESL specification of the MPSoC is automatically derived from the C program. This specification is divided into three XML files: application, platform and mapping.

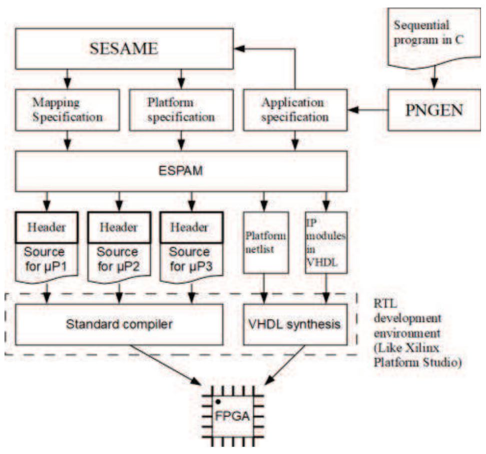


Fig. 2. Daedalus Framework for high-level system synthesis.

Application specification – contains a parallel equivalent of the sequential C program. The application is presented as a set of tasks that exchange data between each other. The model of computation being used is the polyhedral process network or PPN, originally proposed at Leiden University. In it, tasks are concurrent and transfer data through FIFO channels.

Platform specification – describes the topology of the multiprocessor system as a set of processing elements (PE), buses and switches. Memories are also used for the application code and the FIFO buffers.

Mapping specification – describes the link between each application task and each processing element. Simply put, the file tells the system which code is executed on which processor.

The sequential C program must be written as a parameterized static affine nested loop program (SANLP). This is the input expected by the parallelizing tool called PNgen. If not used, the user must write the application specification by hand in XML.

The platform and mapping specification are derived at ESL by a tool called SESAME that can optimize the system for a specific parameter (currently time, cost and power are supported). The optimization process is called design space exploration, or DSE. The user could skip this tool also, in that case an Eclipse plug-in has been developed that loads a GUI editor for mapping and platform, and those specifications could be done by hand (or semi-automatically). SESAME uses a component library that separates entities at two levels of abstraction - high-level for ESL synthesis and modeling of multiprocessor systems, and low-level RTL models to make a transition between the ESL and the RTL designs. As input the SESAME accepts the XML description of the application. When all of the XML files are ready, they are passed to a tool called ESPAM that automatically generates, in several steps, an RTL specification that includes hardware and software. The RTL specification is then fed as an input to a commercial tool that will carry on with the development. Currently the supported IDE is Xilinx (XPS, XSDK and Vivado). The derived files are actually a VHDL description of the MPSoC and C/C++ firmware for the microprocessors. The VHDL is divided in three parts:

– platform topology – a netlist of the MPSoC describing in greater detail the connections between the components;

– hardware descriptions of IP cores – predefined or custom intellectual property (IP) cores such as processors, memories, buses, etc.;

– custom IP cores – auxiliary cores needed as glue logic between the components in the system.

The C/C++ firmware is a low-level representation of the ESL application specification. It contains code for the functional behavior, as well as synchronization of the communication between the PE. The C/C++ source files are input to a common cross compiler, currently GCC ported for Xilinx's Microblaze microprocessor.

An important feature of the Daedalus framework is that the mapping of FIFO channels to memories is not part of the mapping specification. A FIFO channel X is always mapped to a local memory of processing component Y, if the process that writes to X is mapped on processing component Y. Following this rule, ESPAM explicitly derives the mapping of FIFO channels to memories.

III. History of the Daedalus Framework

The origin of the Daedalus framework dates back to the year 2004 when work on key features has started. Two university professors and a PhD student were involved – Ed Deprettere, Todor Stefanov and Hristo Nikolov. All of them worked at Leiden University, more specifically the computer science institute – LIACS. The project is actually a spin off from the commercial tool Compaan. An open-source back-end was added to it, namely ESPAM. To make the entire project open-source, the Compaan was replaced with PNgen. Compaan continues to be a closed source alternative and is used by CompaanDesign BV (https://www.compaandesign.com), founded by Bart Kienhuis and Ed Deprettere. The PNgen was developed in 2005-2006 by Sven Verdoolaege, Todor Stefanov and Hristo Nikolov, and the year 2006 may be considered as the birth year of Daedalus. In the period 2007–2008 along with help from the University of Amsterdam, the SESAME tool had been integrated. Thusly the first full version of the framework was presented in 2008 at the 45th ACM/IEEE Int. Design Automation Conference (DAC'08), Anaheim, USA. To help proliferate the new software, in 2009 the Daedalus foundation was created by Todor Stefanov, Andy Pimentel and Ed Deprettere. In the beginning of 2010, Todor Stefanov and Ed Deprettere from Leiden University and Angel Popov, Marin Marinov and other colleagues from TU Sofia founded the DAEDALUS research and education laboratory, a joint laboratory between LIACS, Leiden University and the Department of Electronics, Faculty of Electronic Engineering and Technology, Technical University of Sofia, Bulgaria.

IV. The Polyhedral Process Network Model of Computation

An integral part of the Daedalus tool flow is the polyhedral process network. As mentioned before, streaming data applications are the target of the current framework. Examples of such uses are in the multimedia, imaging, and signal processing. A polyhedral process network (PPN) is a network of parallel executing tasks that communicate over bounded (restricted in size) FIFO buffers [3]–[5]. Each channel serves streams of data tokens. There are two types of tasks – a producer and a consumer. For each FIFO there is a single producer and a single consumer of data. Multiple producers cannot communicate over a single channel and the same goes for the consumers. The synchronization of the communication is done with a blocking mechanism. If a FIFO buffer is empty, or in other words – no data tokens are stored in it, a read on this FIFO will stall or block the reader until a producer writes some data in it. If a FIFO is full, or in other words – all the registers contain data, a write to this FIFO will make the writer to stall or block until a consumer reads some data from it. Reading from a FIFO is destructive which means that data is removed from the FIFO once it has been read. Reading two times the same value is not possible. At any given clock cycle, a process is either performing calculations or is blocked on some of its channels. A process may exchange data only on one channel at a time. If a process is blocked on some of its channels, it cannot access other channels that are not empty.

An example PPN is shown in Fig. 3. It contains three processes (tasks) that communicate through four FIFO channels. For each process, there is a single microprocessor implemented on the FPGA, hence the multiple main( ) functions. Read and write primitives have the same implementation on all cores, but each core has a library in its local memory that contains copies of the read( ) and write( ) functions. The PPN is a derivative of the more general Kahn process network (KPN). PPN processes go through three phases, namely read, execute, and write. The name polyhedral stems from the behavior of the process in a PPN and it resembles parametrized polyhedral descriptions using the polytope model. Formal descriptions are expressed in the following form

$$D(p) = \left\{ x \in Z^d \,\middle|\, A \times x \geq B \times p + b \right\}, \tag{1}$$

where *D(p)* is a parametrized polytope affinely depending on parameter vector p, x is a variable argument from a linear equation, Z is a union of sets of integral solutions to systems of affine inequalities, d is a number corresponding to the number of dimensions used, A and B are static parameters in a linear equation, and b is an Y-axis intercept parameter from a linear equation. If we look at process P2 in Fig. 3, and by obeying certain rules, called the Daedalus rules for SANLP programs (shown in Section V), every function must be enclosed in a for-statement, and by using a few transformations including dependence analysis (to output a Static Single Assignment Codes, SSAC), linearization (to map FIFO buffers to memory with linearly incremental addresses), and FIFO size calculation (by simulation of the program), the number of iterations of the code at line 18 can be found with the expression that describes a two-dimensional polytope

$$D_9(N,M) = \left\{ (i,j) \in Z^2 \,\middle|\, 2 \leq i \leq N 1 \leq j \leq M + i \right\}. \tag{2}$$

The same way, iterations for code line 8 can be calculated using

$$D_8(N,M) = \left\{ (i,j) \in Z^2 \vee 3 \leq i \leq N 1 \leq j \leq M + i \right\}. \tag{3}$$

If we combine all polytopes that describe the process together, we will accurately capture the behaviour of that process. Because all processes are concurrent and their communication is explicit, this makes a very good fit between PPNs and multi-processor systems. The advantages of PPN models of computation for MPSoC can be summarized as:

– design-time analysable – tasks are described as polytopes (or more specifically as polyhedrons) and FIFO sizes can be calculated in advance;

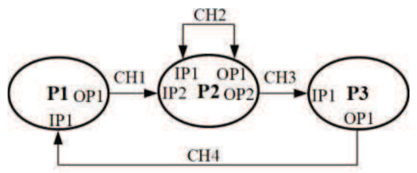
– algebraic transformations can be performed – using mathematical transformations PPNs can be optimized to fit with the processing power of an MPSoC;

– determinism – scheduling of application processes is not crucial and many mappings may exist of one and the same hardware which allows for different types of optimizations;

– distributed control – no global scheduler is needed, scheduling can be done locally with a real-time operating system, or by simply using interrupts if no OS is present;

– distributed memory – no shared memories are being used, thusly avoiding race conditions and inter-processor competition;

– simple synchronization – the process network synchronizes itself due to the blocking read/write mechanism.



```
//Process P1
void main(void){
   for(int k = 1; k <= L; k++){
          read(IP1, in_0, size);
          execute(in_0, out_0);
          write(OP1, out_0, size);
     }
}

//Process P2
void main(void){
   for(int i = 2; i <= N; i++){
   for(int j = 1; j <= M+i; j++){
   if(i – 2 == 0){
          read(IP1, in_0, size);
   }
   if(i – 3 >= 0){
          read(IP2, in_0, size);
   }
   execute(in_0, out_0);
   if(-i+N-1 >= 0){
          write(OP1, out_0, size);
   }
   if(i-N == 0){
          write(OP2, out_0, size);
   }
   }
   }
}
```

Fig. 3. An example of a polyhedral process network and firmware for process P1.

V. The PNgen Tool: Performing Automatic Application Parallelization

As mentioned in the previous sections, the first step in the ESL synthesis starts with automatic transformation of a sequential program in a parallel description. The tool that does this processing is called PNgen and is part of the Daedalus tool flow. The input is a sequential program written in C that complies with static affine nested loop rules. To call a program a static affine nested loop program (SANLP), one must write the program in such that it contains if-statements and function calls enclosed in one or more for-loops. Some rules have to be obeyed:

– loop increments or decrements must be constant;

– loop boundaries must be affine expressions of the enclosing loop iterators, static parameters or constants;

– if statements must have affine conditions of the loop iterators, static parameters or constants;

– static parameters' values may not change during run time;

– function calls must exchange data between each other in an explicit manner, i.e. using only scalar variables, single array elements, and structures (without pointers to other objects);

– array elements must be indexed with affine expressions of the enclosing loop iterators, static parameters or constants.

A program that conforms to those standards is given below and is written in the C programming language:

```
for(int i = 0; i < N; i++){
  b[i] = function_1( );
}

for(int i = 0; i < N; i++){
  if(i > 0){ tmp = b[i - 1]; }
  else{ tmp = b[i]; }
  function_2(b[i], tmp, &c[i]);
}
```

The above-mentioned restrictions allow that the program be represented with the polytope model that uses number sets and integral vectors defined by linear equations and inequations, existential quantification, and the union operation. The set of iterator vectors for which a function call is executed is an integer set that is called iteration domain. The inequality corresponding to this parameter depends on the lower and upper bounds of the for-loop that encloses the function call of interest. For example, the iteration domain of function_1 is $\{i, \mid 0 \le i \le N - 1\}$. All the iteration domains of a program form the basis of the PPN model because each function represents a process. The code given above has two processes that correspond to function_1 and function_2 (shown as ellipses on the graphical representation). FIFO channels are derived from the vector (arrays) or scalar (integers, floating point numbers, etc) accesses by each function call. All of the left-hand side function parameters are considered to be write accesses and must be preceded by the “const” type specifier. The rest parameters on the right-hand side are read accesses and must be “non-const” variables. FIFO channels can be derived using standard array data-flow analysis. This done in the following way: for each read operation by a function call, the respective source of the data has to be found, or in other words – we must find the corresponding function that wrote this data. The answer to this question could be given with the help of parametric integer programming (PIP), where the lexicographical maximum of the write (or read) source operations in terms of the iterators of the “sink” read operation. The PIP operations are performed a number of times depending on the nesting level of the loops. To construct the PPN shown in Fig. 4, we must first note that the first read access in function_2 has read data written by function_1. This results in the FIFO channel represented with an arrow and named “b”. Data flows from iteration i_w of function_1 to i_t of function_2 and can be described as

$$D_{F1\rightarrow F2} = \left\{ \left(i_w, i_r \right) \vee i_r = i_w 0 \le i_r \le N-1 \right\}. \qquad (4)$$

The second read access of function_2 will derive a so-called self-loop channel (shown as b_1 in the figure) because the data has already been read by the same function call after it was written. The temporary variable tmp is eliminated for this assumption. The self-loop channel is described as

$$D_{F2 \to F2} = \left\{ \left(i_w, i_r\right) \middle| i_w = i_r - 1 1 \le i_r \le N - 1 \right\} \cup$$
$$\cup \left\{ \left(i_w, i_r\right) \vee i_w = i_r = 0 \right\}. \tag{5}$$

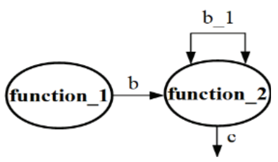


Fig. 4. A graphical representation of a simple SANLP in a PPN.

A union of integer relations could be derived

$$U_{j=1}^{m} D_j \left(i_w, i_r\right) \in Z^{n_1} x Z^{n_2}, \tag{6}$$

where n_1 and n_2 are the number of loops enclosing the write/read and read operations. FIFO channels size have to be calculated in such a way that no deadlocks could occur. The size has to be minimal to save hardware/software resource. The problem could be solved by first computing a deadlock-free schedule. The sizes of each channel can then be calculated individually. This schedule is temporary and not present in the final implementation, its sole purpose is to help with the calculations. The PPN can self-schedule themselves, as mentioned in the previous sections. The intermediate schedule is not guaranteed to be optimal. However, the calculations prove that such a schedule exists for the given buffer sizes. A greedy algorithm is used for the calculations. The basic idea is to place all iteration domains in a common iteration space at an offset that is computed by the scheduling algorithm. The execution order in this common iteration space is the lexicographical order. A minimal dependence distance vector is computed for any pair of connected processes in the application. This vector is actually the difference between a read and the corresponding write operation. All process pairs are greedily combined, in such a way that all distance vectors are positive in lexicographical manner. The end result of this operation ensures that a data element is first written, then read. A loop fusion is also done on the SANLP. When the schedule is complete, all FIFO channels could be considered as self-loops of the common iteration space. For this schedule, the minimum channel sizes can be calculated. For each read iteration *R(i)* that is executed before a given read operation *i* a subtraction is done from the number of write operations *W(i)* preceding the read operation. Therefore, the number of data elements at operation *i*, that depends on the for loop iteration counter, can be expressed as

$$i = W\left(i\right) - R\left(i\right), \tag{7}$$

where W is the write access and R is the read access of that specific iteration. This computation can be done by the readily available Barvinok library. The Barvinok library efficiently computes the number of integer points in a parametric polytope. The output is a polynomial of the read iterators and the parameters. The channel size is the maximum of this output over all read operations

$$\max\left(W\left(i\right) - R\left(i\right)\right). \tag{8}$$

This maximum is calculated with the help of the Bernstein expansion that obtains a parametric upper bound.

VI. The ESPAM Tool: Automated System-Level Synthesis

The tool from the Daedalus design flow that is responsible for automated system-level hardware and software synthesis is called ESPAM (Embedded System-level Platform Synthesis and Application Mapping). It fills the so-called “implementation gap” between the ESL and RTL. A lot of tools exist on those levels but only few can make such a transition in the abstraction. As mentioned before, the input files for ESPAM are application, platform, and mapping.

The *platform specification file* consists of three parts - processing components, communication components and links. There are two ways to create the platform – either edit the platform by hand, using a GUI editor, or use an automated framework, called SESAME [6], to generate it by only giving optimization parameters, e.g. optimize for price, performance, or consumed energy. SESAME also decides the mapping strategy between the abstract description and the platform. An example platform is shown in Fig. 5. Each link connects a processing component to a communication component. Every component has a name and parameters. There are no memory controllers instantiated. ESPAM will automatically handle memories during the synthesis by placing either hardware FIFO buffers implemented on the FPGA, or by placing software FIFOs that are mapped to special communication memories (again implemented on the FPGA). The performance of the memory controllers depends on the RTL library currently being used and is subject to change between revisions. However, most of the memory controllers are as fast as the microprocessor, with frequencies of up to 500 MHz on a Xillinx XUPV5 development board. This is done for the sole purpose of simplification of the design at high level. To be able to generate such a file, the developer needs a library of parametrized components. Such components include 7 types of devices: processing devices, memory, memory controllers, communication components (crossbar switches that connect separate processing elements together), communication controllers (modules that implement the features of a FIFO), peripheral components and links. The processing components, or processing elements (PE), implement the behaviour of the MPSoC by executing code from the process network. The user may choose between programmable processors implementing a certain instruction set and non-programmable dedicated IP cores. Many parameters exist for their configuration such type, number of I/O ports, memory size, etc. Memory models describe either local program and data memories of each processor, or data communication storage used for the FIFO channels. The latter could be mapped onto the local data memory of the PE, or could be implemented with a separate hardware FIFOs. A user should choose only one of the two methods. Some parameters of the memories include type, size, and number of I/O ports. Communication components are used for inter-processor communication and usually crossbar switches are being used (CB in Fig. 5). The topology of the entire MPSoC depends on the connection of the communication component. Currently the Daedalus framework implements two types of topologies: a point-to-point and a multi-FIFO. If at least one communication component is used in the system, it is considered a multi-FIFO topology. If not a single one is used, and PEs are connected directly to each other through hardware FIFOs, then it is considered a point-to-point. Some parameters of the communication component include type and number of I/O ports. Communication controllers (CC) are used for synchronization of the data communication required by the PPN. Memory controllers connect each PE to its respective local memory. Because different types of memories could be used for program and data (SRAM, DRAM, Flash, ROM), different memory controllers also exist. An important memory parameter is the size of the memory. Peripheral components are the entry and exit points of the data to be processed in the streaming data system. Such devices could be UARTs and off-chip memories. For code profiling purposes timer modules are also included. Links are used to connect two or more ports of a device from the MPSoC together. Links are transparent from system-level point of view.

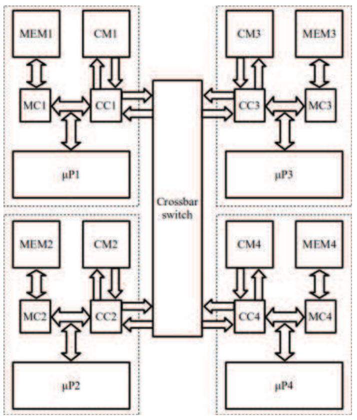


Fig. 5. An example multi-processor platform, where µP is a microprocessor, MC is a memory controller, CC is a communication controller, CM is a communication memory, MEM is a microprocessor's local memory.

The *application specification file* contains a PPN, with processes and FIFO channels in an XML format. This file is the output of the PNgen tool. The information about the number of times a specific function is executed can be written to a parametrized iteration domain captured in a compact matrix form. The application file contains important information about the iterations when an input port has to be read from and when an output port has to be written to.

The *mapping specification file* contains information about the connection between processing elements and application tasks. It is in an XML format and an example is given below. It assumes an MPSoC with four processing components and five PPN processes.

```
<mapping name = "myMapping" >
<processor name = "uP1" >
  <process name = "P4" />
</processor>
<processor name = "uP2" >
  <process name = "P2" />
  <process name = "P5" />
</processor>
<processor name = "uP3" >
  <proces name = "P3" />
</processor>
<processor name = "uP4" >
  <process name = "P1" />
</processor>
</mapping>
```

A single process could be mapped to a single component, like in the case with uP1, uP3 and uP4. Also, multiple processes could be mapped onto a single component, like in uP2. However, a single process cannot be mapped to multiple processing elements which yields an asymmetric multiprocessing system (AMP). The mapping of the FIFO channels to memories is not described anywhere in this file. As mentioned before, this is done by ESPAM automatically.

VII. Conclusions

In the presented paper the authors presented insights about the high-level synthesis framework called Daedalus. It is an open-source framework for ESL synthesis and automatic generation of RTL hardware and software. Modern system design requires that a high-level approach is used to solve the problem and meet the time-to-market deadlines. The framework is complete and ready for production. The future development of Daedalus would be to increase the number of IP cores and supported back-end (RTL) environments. The industry is welcome to enhance and proliferate the tool. The original COMPAAN tool, that spawned Daedalus, has already seen real industrial action [7] and the results are promising.

Acknowledgment

The authors would like to thank the Research and Development Sector at the Technical University of Sofia for the financial support.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] T. Stefanov, E. Deprettere, H. Nikolov, M. Marinov, A. Popov, *Embedded Systems - components, modelling, design and case study*. 2012.

[2] D. D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, *Embedded System Design*. Boston, MA: Springer US, 2009. DOI: 10.1007/978-1-4419-0504-8.

[3] S. Verdoolaege, *"Polyhedral process networks," in Handbook of Signal Processing Systems*. Boston, MA: Springer US, 2010, pp. 931–965. DOI: 10.1007/978-1-4419-6345-1_33.

[4] S. Meijer, H. Nikolov, T. Stefanov, "Combining process splitting and merging transformations for Polyhedral Process Networks", in *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, Scottsdale, AZ, USA, 2010, pp. 97–106. DOI: 10.1109/ESTMED.2010.5666985.

[5] S. Meijer, H. Nikolov, T. Stefanov, "Throughput modeling to evaluate process merging transformations in polyhedral process networks", in *Design, Automation & Test in Europe Conf. & Exhibition (DATE 2010)*, Dresden, Germany, 2010, pp. 747–752. DOI: 10.1109/DATE.2010.5456953.

[6] M. Thompson, A. D. Pimentel, *"Towards multi-application workload modeling in sesame for system-level design space exploration," in Embedded Computer Systems: Architectures, Modeling, and Simulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 222–232. DOI: 10.1007/978-3-540-73625-7_24.

[7] L. Bogdanov, S. Polstra, P. Yakimov, M. Marinov, "DAEDALED: A GUI Tool for the optimization of Smart City LED street lighting networks", in *Proc. XXVII Int. Scientific Conf. Electronics (ET 2018)*, Sozopol, 2018, pp. 125–129.