DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips

Stefanov, T.; Pimentel, A.; Nikolov, H.

DOI
10.1007/978-94-017-7267-9_30
Publication date
2017
Document Version
Final published version
Published in
Handbook of Hardware/Software Codesign
License
Article 25fa Dutch Copyright Act

30 DAEDALUS: System-Level Design Methodology for Streaming Multiprocessor Embedded Systems on Chips

Todor Stefanov, Andy Pimentel, and Hristo Nikolov

Abstract

The complexity of modern embedded systems, which are increasingly based on heterogeneous multiprocessor system-on-chip (MPSoC) architectures, has led to the emergence of system-level design. To cope with this design complexity, system-level design aims at raising the abstraction level of the design process from the register-transfer level (RTL) to the so-called electronic system level (ESL). However, this opens a large gap between deployed ESL models and RTL implementations of the MPSoC under design, known as the *implementation gap*. Therefore, in this chapter, we present the DAEDALUS methodology which the main objective is to bridge this implementation gap for the design of streaming embedded MPSoCs. DAEDALUS does so by providing an integrated and highly automated environment for application parallelization, system-level design space exploration, and system-level hardware/software synthesis and code generation.

Acronyms

ADG	Approximated Dependence Graph
CC	Communication Controller
CM	Communication Memory
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DSE	Design Space Exploration
DWT	Discrete Wavelet Transform
ESL	Electronic System Level
FCFS	First-Come First-Serve

T. Stefanov (✉) • H. Nikolov
Leiden University, Leiden, The Netherlands
e-mail: t.p.stefanov@liacs.leidenuniv.nl; h.n.nikolov@gmail.com

A. Pimentel
University of Amsterdam, Amsterdam, The Netherlands
e-mail: a.d.pimentel@uva.nl

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_30

FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HW	Hardware
IP	Intellectual Property
IPM	Intellectual Property Module
ISA	Instruction-Set Architecture
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Network
MIR	Medical Image Registration
MJPEG	Motion JPEG
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
PIP	Parametric Integer Programming
PN	Process Network
PPN	Polyhedral Process Network
RTL	Register Transfer Level
SANLP	Static Affine Nested Loop Program
STree	Schedule Tree
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language
YML	Y-chart Modeling Language

Contents

30.1 Introduction

The complexity of modern embedded systems, which are increasingly based on heterogeneous multiprocessor system-on-chip (MPSoC) architectures, has led to the emergence of system-level design. To cope with this design complexity, system-level design aims at raising the abstraction level of the design process to the so-called electronic system level (ESL) [18]. Key enablers to this end are, for example, the use of architectural platforms to facilitate reuse of IP components and the notion of high-level system modeling and simulation [21]. The latter allows for capturing the behavior of platform components and their interactions at a high level of abstraction. As such, these high-level models minimize the modeling effort and are optimized for execution speed and can therefore be applied during the very early design stages to perform, for example, architectural design space exploration (DSE). Such early DSE is of paramount importance as early design choices heavily influence the success or failure of the final product.

System-level design for MPSoC-based embedded systems typically involves a number of challenging tasks. For example, applications need to be decomposed into parallel specifications so that they can be mapped onto an MPSoC architecture [29]. Subsequently, applications need to be partitioned into hardware (HW) and software (SW) parts because MPSoC architectures often are heterogeneous in nature. To this end, MPSoC platform architectures need to be modeled and simulated at ESL level of abstraction to study system behavior and to evaluate a variety of different design options. Once a good candidate architecture has been found, it needs to be synthesized. This involves the refinement/conversion of its architectural components from ESL to RTL level of abstraction as well as the mapping of applications onto the architecture. To accomplish all of these tasks, a range of different tools and tool-flows is often needed, potentially leaving designers with all kinds of interoperability problems. Moreover, there typically exists a large gap between the deployed ESL models and the RTL implementations of the system under study, known as the *implementation gap* [32, 37]. Therefore, designers need mature methodologies, techniques, and tools to effectively and efficiently convert ESL system specifications to RTL specifications.

In this chapter, we present the DAEDALUS methodology [27, 37, 38, 40, 51] and its techniques and tools which address the system-level design challenges mentioned above. The DAEDALUS main objective is to bridge the aforementioned implementation gap for the design of streaming embedded MPSoCs. The main idea is, starting with a functional specification of an application and a library of predefined and pre-verified IP components, to derive an ESL specification of an MPSoC and to refine and translate it to a lower RTL specification in a systematic and automated way. DAEDALUS does so by providing an integrated and highly automated environment for application parallelization (Sect. 30.4), system-level DSE (Sect. 30.5), and system-level HW/SW synthesis and code generation (Sect. 30.6).

30.2 The DAEDALUS Methodology

In this section, we give an overview of the DAEDALUS methodology [27, 37, 38, 40, 51]. It is depicted in Fig. 30.1 as a design flow. The flow consists of three main design phases and uses specifications at four levels of abstraction, namely, at FUNCTIONAL-LEVEL, ESL, RTL, and GATE-LEVEL. A typical MPSoC design with DAEDALUS starts at the most abstract level, i.e., with a FUNCTIONAL-LEVEL specification which is an application written as a sequential *C* program representing the required MPSoC behavior. Then, in the first design phase, an ESL specification of the MPSoC is derived from this functional specification by (automated) application parallelization and automated system-level DSE. The derived ESL specification consists of three parts represented in XML format:

1. *Application specification*, describing the initial application in a parallel form as a set of communicating application tasks. For this purpose, we use the polyhedral process network (PPN) model of computation, i.e., a network of concurrent processes communicating via FIFO channels. More details about the PPN model are provided in Sect. 30.3;
2. *Platform specification*, describing the topology of the multiprocessor platform;
3. *Mapping specification*, describing the relation between all application tasks in *application specification* and all components in *platform specification*.

For applications written as parameterized static affine nested loop programs (SANLP) in *C*, a class of programs discussed in Sect. 30.4, PPN descriptions can be derived automatically by using the PNGEN tool [26, 56], see the top-right part in Fig. 30.1. Details about PNGEN are given in Sect. 30.4. By means of automated (polyhedral) transformations [49, 59], PNGEN is also capable of producing alternative input-output equivalent PPNs, in which, for example, the degree of parallelism can be varied. Such transformations enable functional-level design space exploration. In case the application does not fit in the class of programs, mentioned above, the PPN application specification at ESL needs to be derived by hand.

The platform and mapping specifications at ESL are generated automatically as a result of a system-level DSE by using the SESAME tool [8, 39, 42, 53], see the top-left part of Fig. 30.1. Details about SESAME are given in Sect. 30.5. The components in the platform specification are taken from a library of (generic) parameterized and predefined/verified IP components which constitute the platform model in the DAEDALUS methodology. Details about the platform model are given in Sect. 30.6.2. The platform model is a key part of the methodology because it allows alternative MPSoCs to be easily built by instantiating components, connecting them, and setting their parameters in an automated way. The components in the library are

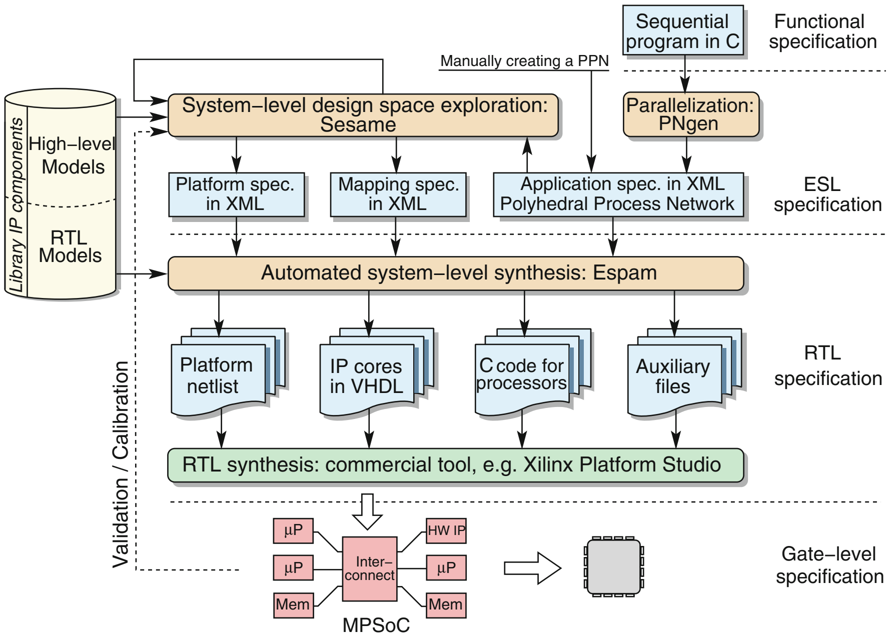


Fig. 30.1 The DAEDALUS design flow

represented at two levels of abstraction: high-level models are used for constructing and modeling multiprocessor platforms at ESL; low-level models of the components are used in the translation of the multiprocessor platforms to RTL specifications ready for final implementation. As input, SESAME uses the application specification at ESL (i.e., the PPN) and the high-level models of the components from the library. The output is a set of pairs, i.e., a platform specification and a mapping specification at ESL, where each pair represents a non-dominated mapping of the application onto a particular MPSoC in terms of performance, power, cost, etc.

In the second design phase, the ESL specification of the MPSoC is systematically refined and translated into an RTL specification by automated system-level HW/SW synthesis and code generation, see the middle part of Fig. 30.1. This is done in several steps by the ESPAM tool [25, 34, 36, 37]. Details about ESPAM are given in Sect. 30.6. As output, ESPAM delivers a hardware (e.g., synthesizable VHDL code) description of the MPSoC and software (e.g., C/C++) code to program each processor in the MPSoC. The hardware description, namely, an RTL specification of a multi-processor system, is a model that can adequately abstract and exploit the key features of a target physical platform at the register-transfer level of abstraction. It consists of two parts: *(1) platform topology*, a netlist description defining in greater detail the MPSoC topology and *(2) hardware descriptions of IP cores*, containing predefined and custom IP cores (processors, memories, etc.) used in *platform topology* and selected from the library of IP components.

Also, ESPAM generates custom IP cores needed as a glue/interface logic between components in the MPSoC. ESPAM converts the application specification at ESL to efficient C/C++ code including code implementing the functional behavior together with code for synchronization of the communication between the processors. This synchronization code contains a memory map of the MPSoC and read/write synchronization primitives. The generated program C/C++ code for each processor in the MPSoC is given to a standard GCC compiler to generate executable code.

In the third and last design phase, a commercial synthesizer converts the generated hardware RTL specification to a GATE-LEVEL specification, thereby generating the target platform gate-level netlist, see the bottom part of Fig. 30.1. This GATE-LEVEL specification is actually the system implementation. In addition, the system implementation is used for validation/calibration of the high-level models in order to improve the accuracy of the design space exploration process at ESL.

Finally, a specific characteristic of the DAEDALUS design flow is that the mapping specification generated by SESAME gives explicitly only the relation between the processes (tasks) in *application specification* and the processing components in *platform specification*. The mapping of FIFO channels to memories is not given explicitly in the mapping specification because, in MPSoCs designed with DAEDALUS, this mapping strictly depends on the mapping of processes to processing components by obeying the following rule. 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. This mapping rule is used by SESAME during the system-level DSE where alternative platform and mapping decisions are explored. The same rule is used by ESPAM (the elaborate mapping step in Fig. 30.7) to explicitly derive the mapping of FIFO channels to memories which is implicit (not explicitly given) in the mapping specification generated by SESAME and forwarded to ESPAM.

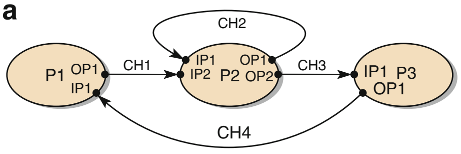
30.3 The Polyhedral Process Network Model of Computation for MPSoC Codesign and Programming

In order to facilitate systematic and automated MPSoC codesign and programming, a parallel model of computation (MoC) is required for the application specification at ESL. This is because the MPSoC platforms contain processing components that run in parallel and a parallel MoC represents an application as a composition of concurrent tasks with a well-defined mechanism for inter-task communication and synchronization. Thus, the operational semantics of a parallel MoC match very well the parallel operation of the processing components in an MPSoC. Many parallel MoCs exist [24], and each of them has its own specific characteristics. Evidently, to make the right choice of a parallel MoC, we need to take into account the application domain that is targeted. The DAEDALUS methodology targets streaming (data-flow-dominated) applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel data-flow MoC called polyhedral process network (PPN) [30, 31, 54]. Therefore, DAEDALUS uses the PPN model as an application specification at ESL as shown in Fig. 30.1.

A PPN is a network of concurrent processes that communicate through *bounded* first-in first-out (FIFO) channels carrying streams of data tokens. A process produces tokens of data and sends them along a FIFO communication channel where they are stored until a destination process consumes them. FIFO communication channels are the only method which processes may use to exchange data. For each channel there is a single process that produces tokens and a single process that consumes tokens. Multiple producers or multiple consumers connected to the same channel are not allowed. The synchronization between processes is done by *blocking on an empty/full FIFO channel*. Blocking on an empty FIFO channel means that a process is suspended when it attempts to consume data from an empty input channel until there is data in the channel. Blocking on a full FIFO channel means that a process is suspended when it attempts to send data to a full output channel until there is room in the channel. At any given point in time, a process either performs some computation or it is blocked on only one of its channels. A process may access only one channel at a time and when blocked on a channel, a process may not access other channels. An example of a PPN is shown in Fig. 30.2a. It consists of three processes ($P1$, $P2$, and $P3$) that are connected through four FIFO channels ($CH1$, $CH2$, $CH3$, and $CH4$).

The PPN MoC is a special case of the more general Kahn process network (KPN) MoC [20] in the following sense. First, the processes in a PPN are uniformly structured and execute in a particular way. That is, a process first reads data from FIFO channels, then executes some computation on the data, and finally writes results of the computation to FIFO channels. For example, consider the PPN shown

a



b

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

c

```
// Process P2
void main( ) {

    for ( int i=2; i<=N; i++ )                                CONTROL
       for ( int j=1; j<=M+i; j++ ) {

             if ( i-2 == 0 )
               read( IP1, in_0, size );                          READ
             if ( i-3 >= 0 )
               read( IP2, in_0, size );

             execute( in_0, out_0 );                          EXECUTE

             if ( -i+N-1 >= 0 )
               write( OP1, out_0, size );                        WRITE
             if ( i-N == 0 )
               write( OP2, out_0, size );

       } // for j
} // main
```

Fig. 30.2 Example of a polyhedral process network and program code of its processes. (**a**) Polyhedral Process Network example. (**b**) Program code for process P1. (**c**) Program code for process P2

in Fig. 30.2a. The program code structure of processes $P1$ and $P2$ are shown in Fig. 30.2b, c, respectively. The structure of the code for both process is the same and consists of a *CONTROL* part, a *READ* part, an *EXECUTE* part, and a *WRITE* part. The difference between $P1$ and $P2$, however, is in the specific code in each part. For example, the *CONTROL* part of $P1$ has only one for loop whereas the *CONTROL* part of $P2$ has two for loops. The blocking synchronization mechanism, explained above, is implemented by read/write synchronization primitives. They are the same for each process. The *READ* part of $P1$ has one read primitive executed unconditionally, whereas the *READ* part of $P2$ has two read primitives and if conditions specifying when to execute these primitives.

Second, the behavior of a process in a PPN can be expressed in terms of parameterized polyhedral descriptions using the polytope model [16], i.e., using formal descriptions of the following form: $D(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Z}^d \mid A \cdot \mathbf{x} \geq B \cdot \mathbf{p} + \mathbf{b}\}$, where $D(\mathbf{p})$ is a parameterized polytope affinely depending on parameter vector $\mathbf{p}$. For example, consider process $P2$ in Fig. 30.2c. The process iterations for which the computational code at line 9 is executed can be expressed as the following two-dimensional polytope: $D_9(N, M) = \{(i, j) \in \mathbb{Z}^2 \mid 2 \leq i \leq N \wedge 1 \leq j \leq M + i\}$. The process iterations for which the read synchronization primitive at line 8 is executed can be expressed as the following two-dimensional polytope: $D_8(N, M) = \{(i, j) \in \mathbb{Z}^2 \mid 3 \leq i \leq N \wedge 1 \leq j \leq M + i\}$. The process iterations for which the other read/write synchronization primitive are executed can be expressed by similar polytopes. All polytopes together capture the behavior of process $P2$, i.e., the code in Fig. 30.2c can be completely constructed from the polytopes and vice versa.

Since PPNs expose task-level parallelism, captured in processes, and make the communication between processes explicit, they are suitable for efficient mapping onto MPSoC platforms. In addition, we motivate our choice of using the PPN MoC in DAEDALUS by observing that the following characteristics of a PPN can take advantage of the parallel resources available in MPSoC platforms:

- **The PPN model is design-time analyzable:** By using the polyhedral descriptions of the processes in a PPN, capacities of the FIFO channels in a PPN, that guarantee deadlock-free execution of the PPN, can be determined at design time;
- **Formal algebraic transformations can be performed on a PPN:** By applying mathematical manipulations on the polyhedral descriptions of the processes in a PPN, the initial PPN can be transformed to an input-output equivalent PPN in order to exploit more efficiently the parallel resources available in an MPSoC platform;
- **The PPN model is determinate:** Irrespective of the schedule chosen to evaluate the network, the same input-output relation always exists. This gives a lot of scheduling freedom that can be exploited when mapping PPNs onto MPSoCs;
- **Distributed Control:** The control is completely distributed to the individual processes and there is no global scheduler present. As a consequence, distributing a PPN for execution on a number of processing components is a relatively simple task;

- **Distributed Memory:** The exchange of data is distributed over FIFO channels. There is no notion of a global memory that has to be accessed by multiple processes. Therefore, resource contention is greatly reduced if MPSoCs with distributed memory are considered;
- **Simple synchronization:** The synchronization between the processes in a PPN is done by a blocking read/write mechanism on FIFO channels. Such synchronization can be realized easily and efficiently in both hardware and software.

Finally, please note that the first and the second bullet, mentioned above, describe characteristics that are specific and valid only for the PPN model. These specific characteristics clearly distinguish the PPN model from the more general KPN model which is used, for example, as an application model in ▶ Chap. 28, "MAPS: A Software Development Environment for Embedded Multicore Applications". The last four bullets above describe characteristics valid for both the PPN and the KPN models.

30.4 Automated Application Parallelization: PNGEN

In this section, we provide an overview of the techniques, we have developed, for automated derivation of PPNs. These techniques are implemented in the PNGEN tool [26, 56] which is part of the DAEDALUS design flow. The input to PNGEN is a SANLP written in *C* and the output is a PPN specification in XML format – see Fig. 30.1. Below, in Sect. 30.4.1, we introduce the SANLPs with their characteristics/limitations and explain how a PPN is derived based on a modified data-flow analysis. We have modified the standard data-flow analysis in order to derive PPNs that have less inter-process FIFO communication channels compared to the PPNs derived by using previous works [23,52]. Then, in Sect. 30.4.2, we explain the techniques to compute the sizes of FIFO channels that guarantee deadlock-free execution of PPNs onto MPSoCs.

30.4.1 SANLPs and Modified Data-Flow Analysis

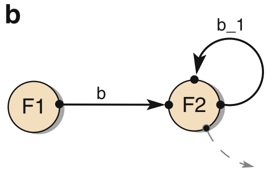
A SANLP is a sequential program that consists of a set of statements and function calls (the code inside function calls is not limited), where each statement and/or function call is possibly enclosed by one or more loops and/or if statements with the following code limitations: (1) loops must have a constant step size; (2) loops must have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants; (3) if statements must have affine conditions in terms of the loop iterators, static program parameters, and constants; (4) the static parameters are symbolic constants, i.e., their values may not change during the execution of the program; (5) the function calls must communicate data between each other explicitly, i.e., using only scalar variables and/or array elements of an arbitrary type that are passed as arguments by value or by reference in function

Fig. 30.3 SANLP fragment and its corresponding PPN. (**a**) Example of a SANLP. (**b**) Corresponding PPN

a

```
for ( int i=0; i<N; i++ )
   b[i] = F1( );
for ( int i=0; i<N; i++ ) {
   if ( i>0 ) tmp = b[i-1];
   else       tmp = b[i];
   F2( b[i], tmp, &c[i] );
}
```

b



calls; (6) array elements must be indexed with affine expressions of the enclosing loop iterators, static program parameters, and constants. An example of a SANLP that conforms to the abovementioned code limitations is shown in Fig. 30.3a. Other examples can be found in [56]. Although the abovementioned code limitations restrict the expressiveness of a SANLP, in many application domains this is not a problem because it is natural to express an application in the form of a SANLP. Examples are digital signal/image processing and audio/video stream-based applications in consumer electronics, medical imaging, radio astronomy, etc. Some specific application examples are mentioned in Sect. 30.7.

Because of the code limitations, mentioned above, SANLPs can be represented in the well-known polytope model [16], i.e., a compact mathematical representation of a SANLP through sets and relations of integral vectors defined by linear (in)equalities, existential quantification, and the union operation. In particular, the set of iterator vectors for which a function call is executed is an integer set called the *iteration domain*. The linear inequalities of this set correspond to the lower and upper bounds of the loops enclosing the function call. For example, the iteration domain of function `F1` in Fig. 30.3a is $\{i \mid 0 \leq i \leq N-1\}$. Iteration domains form the basis of the description of the processes in the PPN model, as each process corresponds to a particular function call. For example, there are two function calls in the program fragment in Fig. 30.3a representing two application tasks, namely, `F1` and `F2`. Therefore, there are two processes in the corresponding PPN as shown in Fig. 30.3b. The granularity of `F1` and `F2` determines the granularity of the corresponding processes. The FIFO channels are determined by the array (or scalar) accesses in the corresponding function call. All accesses that appear on the left-hand side or in an address of (`&`) expression for an argument of a function call are considered to be *write accesses*. All other accesses are considered to be *read accesses*.

To determine the FIFO channels between the processes, we may perform standard array data-flow analysis [15]. That is, for each execution of a read operation of a given data element in a function call, we need to find the source of the data, i.e., the corresponding write operation that wrote the data element. However, to reduce communication FIFO channels between different processes, in contrast to the standard data-flow analysis and in contrast to [23, 52], we also consider all previous read operations from the same function call as possible sources of the data. That is why we call our approach a modified array data-flow analysis [54, 56]. The problem to be solved is then: given a read from an array element, what was the last write to or read from that array element? The last iteration of a function call satisfying some constraints can be obtained by using parametric integer programming (PIP) [14], where we compute the lexicographical *maximum* of the write (or read) source operations in terms of the iterators of the "sink" read operation. Since there may be multiple function calls that are potential sources of the data, and since we also need to express that the source operation is executed before the read (which is not a linear constraint but rather a disjunction of n linear constraints, where n is the shared nesting level), we actually need to perform a number of PIP invocations.

For example, the first read access in function call `F2` of the program fragment in Fig. 30.3a reads data written by function call `F1`, which results in a FIFO channel from process `F1` to process `F2`, i.e., channel `b` in Fig. 30.3b. In particular, data flows from iteration i_w of function `F1` to iteration $i_\mathrm{r} = i_\mathrm{w}$ of function `F2`. This information is captured by the integer relation $D_{\mathtt{F1}\to\mathtt{F2}} = \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{r} = i_\mathrm{w} \wedge 0 \leq i_\mathrm{r} \leq N-1\}$. For the second read access in function call `F2`, after elimination of the temporary variable `tmp`, the data has already been read by the same function call after it was written. This results in a self-loop channel `b_1` from `F2` to itself described as $D_{\mathtt{F2}\to\mathtt{F2}} = \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{w} = i_\mathrm{r} - 1 \wedge 1 \leq i_\mathrm{r} \leq N-1\} \cup \{(i_\mathrm{w}, i_\mathrm{r}) \mid i_\mathrm{w} = i_\mathrm{r} = 0\}$. In general, we obtain pairs of write/read and read operations such that some data flows from the write/read operation to the (other) read operation. These pairs correspond to the channels in our process network. For each of these pairs, we further obtain a union of integer relations $\bigcup_{j=1}^{m} D_j(\mathbf{i}_\mathrm{w}, \mathbf{i}_\mathrm{r}) \subset \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2}$, with n_1 and n_2 the number of loops enclosing the write and read operation, respectively, that connect the specific iterations of the write/read and read operations such that the first is the source of the second. As such, each iteration of a given read operation is uniquely paired off to some write or read operation iteration.

30.4.2 Computing FIFO Channel Sizes

Computing minimal deadlock-free FIFO channel sizes is a nontrivial global optimization problem. This problem becomes easier if we first compute a deadlock-free schedule and then compute the sizes for each channel individually. Note that this schedule is only computed for the purpose of computing the FIFO channel sizes and is discarded afterward because the processes in PPNs are self-scheduled due to the blocking read/write synchronization mechanism. The schedule we compute may not be optimal; however, our computations do ensure that a valid schedule exists for the computed buffer sizes. The schedule is computed using a greedy approach. This approach may not work for process networks in general, but since we consider only static affine nested loop programs (SANLPs), it does work for any PPN derived from a SANLP. The basic idea is to place all iteration domains in a common iteration space at an offset that is computed by the scheduling algorithm. As in the individual iteration spaces, the execution order in this common iteration space is the lexicographical order. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for any pair of connected processes, the minimal dependence distance vector, a distance vector being the difference between a read operation and the corresponding write operation. Then, the processes are greedily combined, ensuring that all minimal distance vectors are (lexicographically) positive. The end result is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [55], where it is applied to perform loop fusion on SANLPs.

After the scheduling, we may consider all FIFO channels to be self-loops of the common iteration space, and we can compute the channel sizes with the following qualification: we will not be able to compute the absolute minimum channel sizes but at best the minimum channel sizes for the computed schedule. To compute the channel sizes, we compute the number of read iterations $R(i)$ that are executed before a given read operation i and subtract the resulting expression from the number of write iterations $W(i)$ that are executed before the given read operation, so the $number\ of\ elements\ in\ FIFO\ at\ operation\ i\ =\ W(i) - R(i)$. This computation can be performed entirely symbolically using the `barvinok` library [57] that efficiently computes the number of integer points in a parametric polytope. The result is a piecewise (quasi-)polynomial in the read iterators and the parameters. The required channel size is the maximum of this expression over all read iterations: $FIFO\ size\ =\ max(\ W(i) - R(i)\ )$. To compute the maximum symbolically, we apply Bernstein expansion [7] to obtain a parametric *upper bound* on the expression.

30.5 Automated System-Level Design Space Exploration: SESAME

In this section, we provide an overview of the methods and techniques we have developed to facilitate automated design space exploration (DSE) for MPSoCs at the electronic system level (ESL). These methods and techniques are implemented in the SESAME tool [8, 11, 39, 42, 53] which is part of the DAEDALUS design flow illustrated in Fig. 30.1. In Sect. 30.5.1, we highlight the basic concept, deployed in SESAME, for system-level DSE of MPSoC platforms. Then, in Sect. 30.5.2, we explain the system-level performance modeling methods and simulation techniques that facilitate the automation of the DSE.

30.5.1 Basic DSE Concept

Nowadays, it is widely recognized that the separation-of-concerns concept [21] is key to achieving efficient system-level design space exploration of complex embedded systems. In this respect, we advocate the use of the popular Y-chart design approach [22] as a basis for (early) system-level design space exploration. This implies that in SESAME, we separate *application models* and *architecture (performance) models* while also recognizing an explicit *mapping step* to map application tasks onto architecture resources. In this approach, an application model – derived from a specific application domain – describes the functional behavior of an application in a timing and architecture independent manner. A (platform) architecture model – which has been defined with the application domain in mind – defines architecture resources and captures their performance constraints. To perform quantitative performance analysis, application models are first mapped onto and then cosimulated with the architecture model under investigation, after which the performance of each application-architecture combination can be evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). Essential in this approach is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, application models can be reused in the exploration cycle. For example, a single application model can be used to exercise different hardware/software partitionings and can be mapped onto a range of architecture models, possibly representing different architecture designs.

30.5.2 System-Level Performance Modeling and Simulation

The SESAME system-level modeling and simulation environment [8, 11, 39, 42, 53] facilitates automated performance analysis of MPSoCs according to the Y-chart design approach as discussed in Sect. 30.5.1, recognizing separate application and architecture models. SESAME has also been extended to allow for capturing power consumption behavior and reliability behavior of MPSoC platforms [44, 45, 50].

The layered infrastructure of SESAME's modeling and simulation environment is shown in Fig. 30.4. SESAME maps application models onto architecture models for cosimulation by means of *trace-driven simulation* while using an intermediate mapping layer for scheduling and event-refinement purposes. This trace-driven simulation approach allows for maximum flexibility and model reuse in the process of exploring different MPSoC configurations and mappings of applications to these MPSoC platforms [8, 11]. To actually explore the design space to find good system implementation candidates, SESAME typically deploys a genetic algorithm (GA). For example, to explore different mappings of applications onto the underlying platform architecture, the mapping of application tasks and inter-task communications can be encoded in a chromosome, which is subsequently manipulated by the genetic operators of the GA [9] (see also ▶ Chap. 9, "Scenario-Based Design Space Exploration"). The remainder of this section provides an overview of each of the SESAME layers as shown in Fig. 30.4.

30.5.2.1 Application Modeling

For application modeling within the DAEDALUS design flow, SESAME uses the polyhedral process network (PPN) model of computation, as discussed in Sect. 30.3, in which parallel processes communicate with each other via bounded FIFO

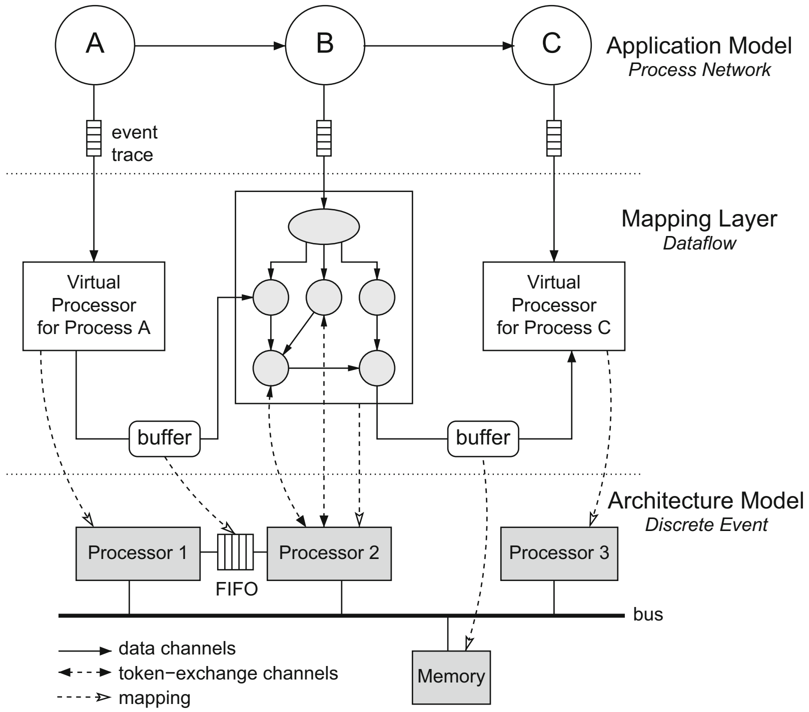


Fig. 30.4 The SESAME's application model layer, architecture model layer, and mapping layer which interfaces between application and architecture models

channels. The PPN application models used in SESAME are either generated by the PNGEN tool presented in Sect. 30.4 or are derived by hand from sequential C/C++ code. The workload of an application is captured by manually instrumenting the code of each PPN process with annotations that describe the application's computational and communication actions, as explained in detail in [8, 11]. By executing the PPN model, these annotations cause the PPN processes to generate traces of *application events* which subsequently drive the underlying architecture model. There are three types of application events: the communication events *read* and *write* and the computational event *execute*. These application events typically are coarse grained, such as *execute(DCT)* or *read(pixel-block,channel_id)*.

To execute PPN application models, and thereby generating the application events that represent the workload imposed on the architecture, SESAME features a process network execution engine supporting the PPN semantics (see Sect. 30.3). This execution engine runs the PPN processes, which are written in C++, as separate threads using the Pthreads package. To allow for rapid creation and modification of models, the structure of the application models (i.e., which processes are used in the model and how they are connected to each other) is not hard-coded in the C++ implementation of the processes. Instead, it is described in a language called

YML (Y-chart modeling language) [8], which is an XML-based language. It also facilitates the creation of libraries of parameterized YML component descriptions that can be instantiated with the appropriate parameters, thereby fostering reuse of (application) component descriptions. To simplify the use of YML even further, a YML editor has also been developed to compose model descriptions using a GUI.

30.5.2.2 Architecture Modeling

The architecture models in SESAME, which typically operate at the so-called transaction level [6, 19], simulate the performance consequences of the computation and communication events generated by an application model. These architecture models solely account for architectural performance constraints and do not need to model functional behavior. This is possible because the functional behavior is already captured in the application models, which subsequently drive the architecture simulation. An architecture model is constructed from generic building blocks provided by a library, see Fig. 30.1, which contains template performance models for processing components (like processors and IP cores), communication components (like busses, crossbar switches, etc.), and various types of memory. The performance parameter values for these models are typically derived from datasheets or from measurements with lower-level simulators or real hardware platforms [43]. The structure of an architecture model – specifying which building blocks are used from the library and the way they are connected – is also described in YML within SESAME.

SESAME's architecture models are implemented using either Pearl [33] or SystemC [19]. Pearl is a small but powerful discrete-event simulation language which provides easy construction of the models and fast simulation [42].

30.5.2.3 Mapping

To map PPN processes (i.e., their event traces) from an application model onto architecture model components, SESAME provides an intermediate *mapping layer*. Besides this mapping function, the mapping layer has two additional functions as will be explained later on: Scheduling of application events when multiple PPN processes are mapped onto a single architecture component (e.g., a programmable processor) and facilitating gradual model refinement by means of trace event refinement.

The mapping layer consists of virtual processor components and FIFO buffers for communication between the virtual processors. There is a one-to-one relationship between the PPN processes in the application model and the virtual processors in the mapping layer. This is also true for the PPN channels and the FIFO buffers in the mapping layer. The only difference is that the buffers in the mapping layer are limited in size, and their size depends on the modeled architecture. As the structure of the mapping layer is equivalent to the structure of the application model under investigation, SESAME provides a tool that is able to automatically generate the mapping layer from the YML description of an application model.

A virtual processor in the mapping layer reads in an application trace from a PPN process via a trace event queue and dispatches the events to a processing component in the architecture model. The mapping of a virtual processor onto a processing component in the architecture model is freely adjustable (i.e., virtual processors can dispatch trace events to any specified processing component in the architecture model), and this mapping is explicitly described in a YML-based specification. Clearly, this YML mapping description can easily be manipulated by design space exploration engines to, e.g., facilitate efficient mapping exploration. Communication channels, i.e., the buffers in the mapping layer, are also explicitly mapped onto the architecture model. In Fig. 30.4, for example, one buffer is placed in shared memory, while the other buffer is mapped onto a point-to-point FIFO channel between processors 1 and 2.

The mechanism used to dispatch application events from a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces [42]. Please note that, here, we refer to communication deadlocks caused by mapping multiple PPN processes to a single processor and the fact that these processes are not preempted when blocked on, e.g., reading from an empty FIFO buffer (see [42] for a detailed discussion of these deadlock situations). In this event dispatching mechanism, computation events are always directly dispatched by a virtual processor to the architecture component onto which it is mapped. The latter schedules incoming events that originate from different event queues according to a given policy (FCFS, round-robin, or customized) and subsequently models their timing consequences. Communication events, however, are not directly dispatched to the underlying architecture model. Instead, a virtual processor that receives a communication event first consults the appropriate buffer at the mapping layer to check whether or not the communication is safe to take place so that no deadlock can occur. Only if it is found to be safe (i.e., for read events the data should be available and for write events there should be room in the target buffer), then communication events may be dispatched. As long as a communication event cannot be dispatched, the virtual processor blocks. This is possible because the mapping layer executes in the same simulation as the architecture model. Therefore, both the mapping layer and the architecture model share the same simulation-time domain. This also implies that each time a virtual processor dispatches an application event (either computation or communication) to a component in the architecture model, the virtual processor is blocked in simulated time until the event's latency has been simulated by the architecture model. In other words, the individual virtual processors can be seen as abstract representations of application processes at the system architecture level, while the mapping layer can be seen as an abstract OS model.

When architecture model components need to be gradually refined to disclose more implementation details (such as pipelined processing in processor components), this typically implies that the applications events consumed by the architecture model also need to be refined. In SESAME, this is established by an approach in which the virtual processors at the mapping layer are also refined. The latter is done by incorporating data-flow graphs in virtual processors such that it allows us to perform architectural simulation at multiple levels of abstraction without modifying the application model. Fig. 30.4 illustrates this data-flow-based refinement by refining the virtual processor for process B with a fictive data-flow graph. In this approach, the application event traces specify *what* a virtual processor executes and *with whom* it communicates, while the internal data-flow graph of a virtual processor specifies *how* the computations and communications take place at the architecture level. For more details on how this refinement approach works, we refer the reader to [10,12,41] where the relation between event trace transformations for refinement and data-flow actors at the mapping layer is explained.

30.6 Automated System-Level HW/SW Synthesis and Code Generation: ESPAM

In this section, we present the methods and techniques, we have developed, for systematic and automated system-level HW/SW synthesis and code generation for MPSoC design and programming. These methods and techniques bridge, in a particular way, the *implementation gap* between the electronic system level (ESL) and the register transfer level (RTL) of design abstraction introduced in Sect. 30.1. The methods and techniques are implemented in the ESPAM tool [25, 34, 36, 37] which is part of the DAEDALUS design flow illustrated in Fig. 30.1 and explained in Sect. 30.2. First, in Sect. 30.6.1, we show an example of the ESL input specification for ESPAM that describes an MPSoC. Second, in Sect. 30.6.2, we introduce the system-level platform model used in ESPAM to construct MPSoC platform instances at ESL. Then, in Sect. 30.6.3, we present how an MPSoC platform instance at ESL is refined and translated systematically and automatically to an MPSoC instance at RTL. This is followed by a discussion in Sect. 30.6.4 about the automated programming of the MPSoCs, i.e., the automated code generation done by ESPAM. It includes details on how ESPAM converts processes in a PPN application specification to software code for every programmable processor in an MPSoC. Finally, in Sect. 30.6.5, we present our approach for building heterogeneous MPSoCs where both programmable processors and dedicated IP cores are used as processing components.

30.6.1 ESL Input Specification for ESPAM

Recall from Sect. 30.2 that ESPAM requires as input an ESL specification of an MPSoC that consists of three parts: *platform, application, and mapping specifications*. In this section, we give examples of these three parts (specifications). We will use these examples in our discussion about the system-level HW/SW synthesis and code generation in ESPAM given in Sects. 30.6.3 and 30.6.4.

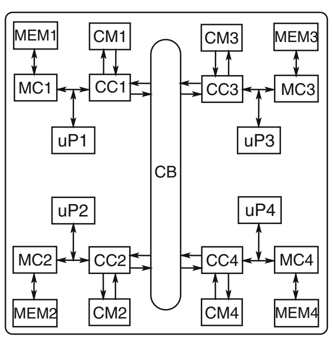
30.6.1.1 Platform Specification

Consider an MPSoC platform containing four processing components. An example of the ESL platform specification of this MPSoC is depicted in Fig. 30.5a. This specification, in XML format, consists of three parts which define processing components (four processors, lines 2–5), communication component (crossbar, lines

a

```
 1 <platform name = "myPlatform" >
 2    <processor name = "uP1" > <port name = "IO1" /> </processor>
 3    <processor name = "uP2" > <port name = "IO1" /> </processor>
 4    <processor name = "uP3" > <port name = "IO1" /> </processor>
 5    <processor name = "uP4" > <port name = "IO1" /> </processor>
 6
 7    <network  name = "CB"   type = "Crossbar" >
 8         <port  name = "IO1" />
 9         <port  name = "IO2" />
10         <port  name = "IO3" />
11         <port  name = "IO4" />
12    </network>
13
14    <link  name = "BUS1"  >
15         <resource  name = "CB"    <port  name = "IO1" />
16         <resource  name = "uP1"   <port  name = "IO1" />
17    </link>
18    <link  name = "BUS2"  >
19         <resource  name = "CB"    <port  name = "IO2" />
20         <resource  name = "uP2"   <port  name = "IO1" />
21    </link>
22    <link  name = "BUS3"  >
23         <resource  name = "CB"    <port  name = "IO3" />
24         <resource  name = "uP3"   <port  name = "IO1" />
25    </link>
26    <link  name = "BUS4"  >
27         <resource  name = "CB"    <port  name = "IO4" />
28         <resource  name = "uP4"   <port  name = "IO1" />
29    </link>
30 </platform>
```

b



Legend:

uP	– Microprocessor
MC	– Memory Controller
MEM	– Program and Data Memory
CC	– Communication Controller
CM	– Communication Memory
CB	– Crossbar Switch

Fig. 30.5 Example of a multiprocessor platform. (**a**) Platform specification. (**b**) Elaborated platform

7–12), and links (lines 14–29). The links specify the connections of the processors to the communication component. Every component has an instance name and different parameters characterizing the component. The components' parameters are not shown in Fig. 30.5a for the sake of brevity. Note that in the specification, there are no memory structures and interface controllers instantiated. The ESPAM tool takes care of this during the platform synthesis described in Sect. 30.6.3. In this way, unnecessary details are hidden at the beginning of the design, keeping the abstraction of the input platform specification very high.

30.6.1.2 Application Specification

Consider an application specified as a PPN consisting of five processes communicating through seven FIFO channels. A graphical representation of the application is shown in Fig. 30.9a. Part of the corresponding XML application specification for this PPN is shown in Fig. 30.6a. Recall that this PPN in XML format is generated automatically by the PNGEN tool using the techniques presented in Sect. 30.4. For the sake of clarity, we show only the description of process $P1$ and channel $FIFO1$ in the XML code. The other processes and channels of the PPN are specified in an identical way. $P1$ has one input port and one output port defined in lines 3–8. $P1$ executes a function called $compute$ (line 9). The function has one input argument (line 10) and one output argument (line 11). There is a strong relation between the function arguments and the ports of a process given at lines 4 and 7. The information how many times function $compute$ has to be fired during the execution of the application is determined by a parameterized *iteration domain* (see

a

```
1  <application name = "myPPN" >
2    <process name = "P1" >
3      <port name = "p2"  direction = "in"  >
4          <var name = "in_0"  type = "myType"  />
5      </port>
6      <port name = "p1"  direction = "out"  >
7          <var name = "out_0"   type = "myType"  />
8      </port>
9      <process_code  name = "compute"  >
10         <arg name = "in_0"  type = "input" />
11         <arg name = "out_0"   type = "output"  />
12         <loop  index = "k"   parameter = "N"  >
13           <loop_bounds  matrix = "[1,  1,0,–2;  1,–1,2,–1]"/>
14           <par_bounds   matrix = "[1,0,–1,384;  1,0,1,–3]"/>
15         </loop>
16     </process_code>
17   </process>
18         <!–– other processes omitted ––>
19   <channel name = FIFO1 size = "1"  >
20       <fromPort  name = "p1" />
21       <fromProcess  name = "P1"/>
22       <toPort  name = "p4" />
23       <toProcess  name = "P3"/>
24    </channel>
25         <!–– other channels omitted ––>
26 </application>
```

b

```
1  <mapping  name = "myMapping" >
2
3    <processor  name = "uP1"  >
4         <process  name = "P4" />
5    </processor>
6
7    <processor  name = "uP2"  >
8         <process  name = "P2" />
9         <process  name = "P5" />
10    </processor>
11
12    <processor  name = "uP3"  >
13         <proces  name = "P3" />
14    </processor>
15
16    <processor  name = "uP4"  >
17         <process  name = "P1" />
18    </processor>
19
20 </mapping>
```

Fig. 30.6 Example of application and mapping specifications. (**a**) Application specification. (**b**) Mapping specification

Sect. 30.4.1) which is captured in a compact (matrix) form at lines 12–15. There are two matrices representing the iteration domain which corresponds to a nested *for*-loop structure. It originates from the structure of the initial (static and affine) nested loop program. In this particular example, there is only one *for* loop with index k and parameter N. The parameter is used in determining the upper bound of the loop. The range of the loop index k is determined at line 13. This matrix represents the following two inequalities, $k - 2 \geq 0$ and $-k + 2N - 1 \geq 0$ and, therefore, $2 \leq k \leq 2N - 1$. In the same way, the matrix at line 14 determines the range of parameter N, i.e., $3 \leq N \leq 384$. Similar information for each port is used to determine at which iterations an input port has to be read and consequently, at which iterations, an output port has to be written. However, for brevity, this information is omitted in Fig. 30.6a. Lines 19–24 show an example of how the topology of a PPN is specified: $FIFO1$ connects processes $P1$ and $P3$ through ports $p1$ and $p4$.

30.6.1.3 Mapping Specification

An example of a mapping specification is shown in Fig. 30.6b. It assumes an MPSoC with four processing components, namely, $uP1$, $uP2$, $uP3$, and $uP4$, and five PPN processes: $P1$, $P2$, $P3$, $P4$, and $P5$. The XML format of the mapping specification is very simple. Process $P4$ is mapped onto processor $uP1$ (see lines 3-5), processes $P2$ and $P5$ are mapped onto processor $uP2$ (lines 7-10), process $P3$ is mapped for execution on processor $uP3$, and, finally, process $P1$ is mapped on processor $uP4$. In the mapping specification, the mapping of FIFO channels to communication memories is not specified. This mapping is related to the way processes are mapped to processors, and, therefore, the mapping of FIFO channels to communication memories cannot be arbitrary. The mapping of channels is performed by ESPAM automatically which is discussed in Sect. 30.6.3.

30.6.2 System-Level Platform Model

The platform model consists of a library of generic parameterized components and defines the way the components can be assembled. To enable efficient execution of PPNs with low overhead, the platform model allows for building MPSoCs that strictly follow the PPN operational semantics. Moreover, the platform model allows easily to construct platform instances at ESL. To support systematic and automated synthesis of MPSoCs, we have carefully identified a set of components which comprise the MPSoC platforms we consider. It contains the following components.

Processing Components. The processing components implement the functional behavior of an MPSoC. The platform model supports two types of processing components, namely, programmable (ISA) processors and non-programmable, dedicated IP cores. The processing components have several parameters such as *type*, *number of I/O ports*, program and data *memory size*, etc.

Memory Components. Memory components are used to specify the local program and data memories of the programmable processors and to specify data communication storage (buffers) between the processing components (*communication memories*). In addition, the platform model supports dedicated FIFO components used as communication memories in MPSoCs with a point-to-point topology. Important memory component parameters are *type*, *size*, and *number of I/O ports*.

Communication Components. A communication component determines the interconnection topology of an MPSoC platform instance. Some of the parameters of a communication component are *type* and *number of I/O ports*.

Communication Controller. Compliant with our approach to build MPSoCs executing PPNs, communication controllers are used as glue logic realizing the synchronization of the data communication between the processors at hardware level. A communication controller (CC) implements an interface between processing, memory, and communication components. There are two types of CCs in our library. In case of a point-to-point topology, a CC implements only an interface to the dedicated FIFO components used as communication memories. If an MPSoC utilizes a communication component, then the communication controller realizes a multi-FIFO organization of the communication memories. Important CC parameters are *number of FIFOs* and the *size* of each FIFO.

Memory Controllers. Memory controllers are used to connect the local program and data memories to the ISA processors. Every memory controller has a parameter *size* which determines the amount of memory that can be accessed by a processor through the memory controller.

Peripheral Components and **Controllers**. They allow data to be transferred in and out of the MPSoC platform, e.g., a universal asynchronous receive-transmit

(UART). We have also developed a multi-port interface controller allowing for efficient (DMA-like) data communication between the processing cores by sharing an off-chip memory organized as multiple FIFO channels [35]. General off-chip memory controller is also part of this group of library components. In addition, *Timers* can be used for profiling and debugging purposes, e.g., for measuring execution delays of the processing components.

Links. Links are used to connect the components in our system-level platform model. A link is transparent, i.e., it does not have any type, and connects ports of two or more components together.

In DAEDALUS we do not consider the design of processing components. Instead, we use IP cores (programmable processors and dedicated IPs) developed by third parties and propose a communication mechanism that allows efficient data communication (low latency) between these processing components. The devised communication mechanism is independent of the types of processing and communication components used in the platform instance. This results in a platform model that easily can be extended with additional (processing, communication, etc.) components.

30.6.3 Automated System-Level HW Synthesis and Code Generation

The automated translation of an ESL specification of an MPSoC (see Sect. 30.6.1 for an example of such specification) to RTL descriptions goes in three main steps illustrated in Fig. 30.7:

1. **Model initialization.** Using the platform specification, an MPSoC instance is created by initializing an abstract platform model in ESPAM. Based on the application and the mapping specifications, three additional abstract models are initialized: application (ADG), schedule (STree), and mapping models;
2. **System synthesis.** ESPAM elaborates and refines the abstract platform model to a detailed parameterized platform model. Based on the application, schedule, and mapping models, a parameterized process network (PN) model is created as well;
3. **System generation.** Parameters are set and ESPAM generates a platform instance implementation using the RTL version of the components in the library. In addition, ESPAM generates program code for each programmable processor.

30.6.3.1 Model Initialization

In this first step, ESPAM constructs a platform instance from the input platform specification by initializing an abstract platform model. This is done by instantiating and connecting the components in the specification using abstract components from the library. The abstract model represents an MPSoC instance without taking target execution platform details into account. The model includes key system components and their attributes as defined in the platform specification. There are three additional abstract models in ESPAM which are also created and initialized, i.e., an application model, a schedule model, and a mapping model, see the top part

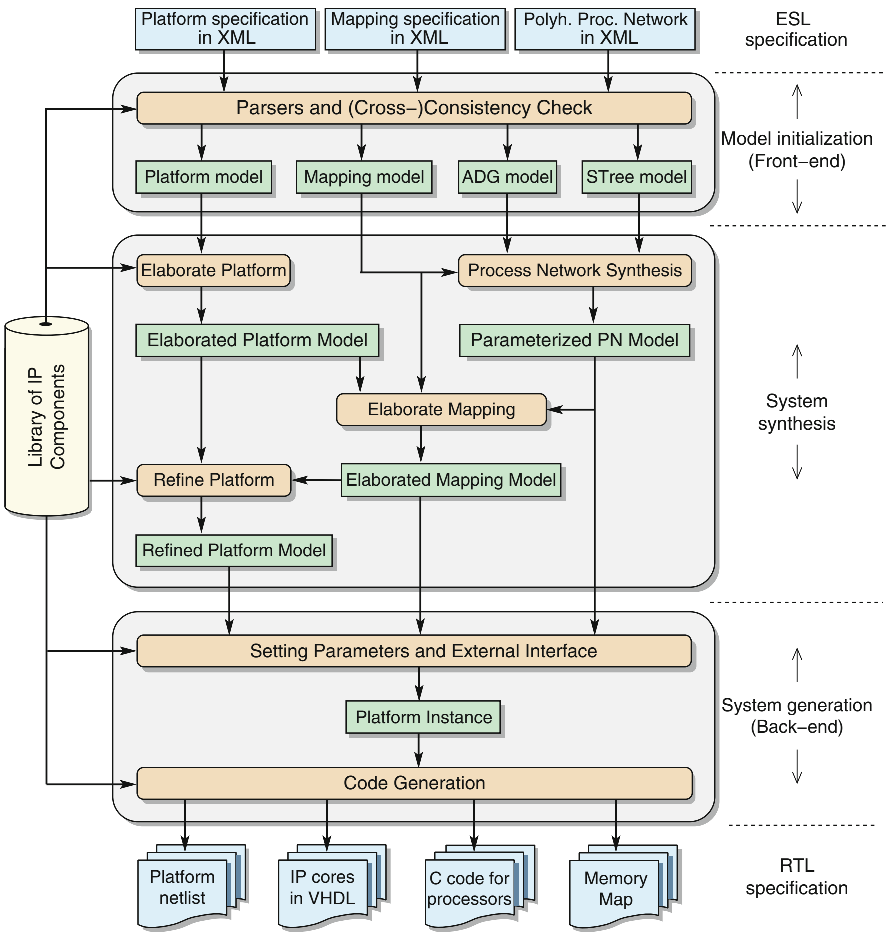


Fig. 30.7 ESL to RTL MPSoC synthesis steps performed by ESPAM

of Fig. 30.7. The application specification consists of two annotated graphs, i.e., a PPN represented by an *approximated dependence graph* (ADG) and a *schedule tree* (STree) representing one valid global schedule of the PPN. Consequently, the ADG and the STree models in ESPAM are initialized, capturing in a formal way all the information that is present in the application specification. Note that, in addition to a standard dependence graph, the ADG is a graph structure that also can capture some data dependencies in an application that are not completely known at design time because the exact application behavior may depend on the data that is processed by the application at run time. If such application is given to ESPAM where some of the data dependencies cannot be exactly determined at design time, then these dependencies are approximated in the ADG. That is, these dependencies are always conservatively put in the ADG, although they may exist only for specific data values processed by the application at run time.

The mapping model is constructed and initialized from the mapping specification. The objective of the mapping model in ESPAM is to capture the relation between the PPN processes in an application and the processing components in an MPSoC instance on the one hand and the relation between FIFO channels and communication memories on the other. The mapping model in ESPAM contains important information which enables the generation of the memory map of the system in an automated way – see Sect. 30.6.4.

30.6.3.2 System Synthesis

The system synthesis step is comprised of several sub-steps. These are platform and mapping model elaboration, process network (PN) synthesis, and platform instance refinement sub-steps. As a result of the platform elaboration, ESPAM creates a detailed parameterized model of a platform instance – see an example of such elaborated platform instance in Fig. 30.5b. The details in this model come from additional components added by ESPAM in order to construct a complete system. In addition, based on the type of the processors instantiated in the first step, the tool automatically synthesizes, instantiates, and connects all necessary communication controllers (CCs) and communication memories (CMs). After the elaboration, a refinement (optimization) step is applied by ESPAM in order to improve resource utilization and efficiency. The refinement step includes program and data memory refinement and compaction in case of processing components with RISC architecture, memory partitioning, and building the communication topology in case of point-to-point MPSoCs. As explained at the end of Sect. 30.2, the mapping specification generated by SESAME contains the relation between PPN processes and processing components only. The mapping of FIFO channels to memories is not given explicitly in the mapping specification. Therefore, ESPAM derives automatically the mapping of FIFO channels to communication memories. This is done in the mapping elaboration step, in which the mapping model is analyzed and augmented with the mapping of FIFO channels to communication memories following the mapping rule described in Sect. 30.2. The PN synthesis is a translation of the *approximated dependence graph* (ADG) model and the *schedule tree* (STree) model into a (parameterized) process network model. This model is used for automated SW synthesis and SW code generation discussed in Sect. 30.6.4.

30.6.3.3 System Generation

This final step consists of a setting parameters sub-step which completely determines a platform instance and a code generation sub-step which generates hardware and software descriptions of an MPSoC. In ESPAM, a software engineering technique called *Visitor* [17] is used to visit the PN and platform model structures and to generate code. For example, ESPAM generates VHDL code for the HW part, i.e., the HW components present in the platform model by instantiating components' templates written in VHDL which are part of the library of IP components. Also, ESPAM generates C/C++ code for the SW part captured in the PN model. The automated SW code generation is discussed in Sect. 30.6.4. The HW description generated by ESPAM consists of two parts: **(1) Platform topology.** This is a netlist description defining the MPSoC topology that corresponds to the platform instance synthesized by ESPAM. This description contains the components of the platform instance with the appropriate values of their parameters and the connections between the components in the form compliant with the input requirements of the commercial tool used for low-level synthesis. **(2) Hardware descriptions of the MPSoC components.** To every component in the platform instance corresponds a detailed description at RTL. Some of the descriptions are predefined (e.g., processors, memories, etc.), and ESPAM selects them from the library of components and sets their parameters in the platform netlist. However, some descriptions are generated by ESPAM, e.g., an IP Module used for integrating a third-party IP core as a processing component in an MPSoC (discussed in Sect. 30.6.5).

30.6.4 Automated System-Level SW Synthesis and Code Generation

In this section, we present in detail our approach for systematic and automated programming of MPSoCs synthesized with ESPAM. For the sake of clarity, we explain the main steps in the ESPAM programming approach by going through an illustrative example considering the input platform, application, and mapping specifications described in Sect. 30.6.1. For these example specifications, we show how the SW code for each processor in an MPSoC platform is generated and present our SW synchronization and communication primitives inserted in the code. Finally, we explain how the memory map of the MPSoC is generated.

30.6.4.1 SW Code Generation for Processors

ESPAM uses the initial sequential application program, the corresponding PPN application specification, and the mapping specification to generate automatically software (C/C++) code for each processor in the platform specification. The code for a processor contains $control$ code and $computation$ code. The $computation$ code transforms the data that has to be processed by a processor, and it is grouped into function calls in the initial sequential program. ESPAM extracts this code directly from the sequential program. The $control$ code (for loops, if statements, etc.) determines the control flow, i.e., when and how many times data reading and data writing have to be performed by a processor as well as when and how many times the $computation$ code has to be executed in a processor. The $control$ code of a processor is generated by ESPAM according to the PPN application specification and the mapping specification as we explain below.

According to the mapping specification in Fig. 30.6b, process $P1$ is mapped onto processor $uP4$ (see lines 16–18). Therefore, ESPAM uses the XML specification of process $P1$ shown in Fig. 30.6a to generate the $control$ C code for processor $uP4$. The code is depicted in Fig. 30.8a. At lines 4–7, the type of the data transferred through the FIFO channels is declared. The data type can be a scalar or more complex type. In this example, it is a structure of 1 Boolean variable and a 64-element array of integers, a data type found in the initial sequential program. There is one parameter (N) that has to be declared as well. This is done at line 8

a

```
 1 #include "primitives.h"
 2 #include "memoryMap.h"
 3
 4 struct myType {
 5     bool flag;
 6     int  data[64];
 7 };
 8 int N = 384;
 9
10 void main( ) {
11     myType in_0;
12     myType out_0;
13
14     for ( int k=2; k<=2*N−1; k++ ) {
15         read( p2, &in_0, sizeof(myType) );
16         compute( in_0, &out_0 );
17         write( p1, &out_0, sizeof(myType) );
18     }
19 }
```

b

```
 1 void read( int* port, void* data, int length ) {
 2     int *req_&_rd = 0xE0000000; // Address in a CC
 3     int *isEmpty = req_&_rd + 1;
 4     *req_&_rd = 0x80000000 | (port);  // Write a request
 5     for ( int i=0; i<length; i++ ) {
 6         // reading is blocked if a FIFO is empty
 7         while ( *isEmpty ) { }
 8         (byte* data)[i] = *req_&_rd; // read from a FIFO
 9     }
10     *req_&_rd = 0x7FFFFFFF&(inPort);
11 }
12
13 void write( int* port, void* data, int length ) {
14     int *isFull = port + 1;
15     for ( int i=0; i<length; i++ ) {
16         // writing is blocked if a FIFO is full
17         while ( *isFull ) { }
18         *port = (byte* data)[i]; // write to a FIFO
19     }
20 }
```

Fig. 30.8 Source code generated by ESPAM. (**a**) Control code for processor $uP4$. (**b**) Read and write communication primitives

in Fig. 30.8a. Then, at lines 10–19 in the same figure, the behavior of processor $uP4$ is described. In accordance with the XML specification of process $P1$ in Fig. 30.6a, the function $compute$ is executed $2 * N - 2$ times. Therefore, a *for* loop is generated in the $main$ routine for processor $uP4$ in lines 14-18 in Fig. 30.8a. The $computation$ code in function $compute$ is extracted from the initial sequential program. This code is not important for our example; hence, it is not given here for the sake of brevity. The function $compute$ uses local variables in_0 and out_0 declared in lines 11 and 12 in Fig. 30.8a. The input data comes from $FIFO2$ through port $p2$, and the results are written to $FIFO1$ through port $p1$ – see Fig. 30.9a. Therefore, before the function call, ESPAM inserts a *read* primitive to read from $FIFO2$ initializing variable in_0 and after the function call, ESPAM inserts a *write* primitive to send the results (the value of variable out_0) to $FIFO1$ as shown in Fig. 30.8a at lines 15 and 17. When several processes are mapped onto one processor, a schedule is required in order to guarantee a proper execution order of these processes onto one processor. The ESPAM tool automatically finds a local static schedule from the STree model (see Sect. 30.6.3) based on the grouping technique for processes presented in [48].

30.6.4.2 SW Communication and Synchronization Primitives

Recall from Sect. 30.6.2 that the FIFO channels are mapped onto the communication memories of an MPSoC platform instances and the multi-FIFO organization of a communication memory is realized by the corresponding communication controller (CC). A FIFO channel is seen by a processor as two memory locations in its communication memory address space. A processor uses the first location to read the status of the FIFO. The status indicates whether a FIFO is full (data cannot be written) or empty (data is not available). This information is used for the inter-processor synchronization. The second location is used to read/write data from/to the FIFO buffer, thereby, realizing inter-processor data transfer.

The described behavior is realized by the SW communication and synchronization primitives interacting with the HW communication controllers. The code implementing the *read* and *write* primitives used in lines 15 and 17 in Fig. 30.8a is shown in Fig. 30.8b. Both read and write primitives have three parameters: $port$, $data$, and $length$. Parameter $port$ is the address of the memory location through which a processor can access a given FIFO channel for reading/writing. Parameter $data$ is a pointer to a local variable and $length$ specifies the amount of data (in bytes) to be moved from/to the local variable to/from the FIFO channel. The primitives implement the blocking synchronization mechanism between the processors in the following way. First, the status of a channel that has to be read/written is checked. A channel status is accessed using the locations defined in lines 3 and 14. The blocking is implemented by while loops with empty bodies (busy-polling mechanism) in lines 7 and 17. A loop iterates (does nothing) while a channel is full or empty. Then, in lines 8 and 18 the actual data transfer is performed. Note that the busy-polling mechanism, described above, to implement the blocking is sufficient because PPN processes mapped onto a processor are statically scheduled, and the busy-polling mechanism exactly follows/implements the blocking semantics of a PPN process, discussed in the second paragraph of Sect. 30.3, thereby guaranteeing deterministic execution of the PPN.

30.6.4.3 Memory Map Generation

Each FIFO channel in an MPSoCs has separate read and write ports. A processor accesses a FIFO for read operations using the *read* synchronization primitive. The parameter $port$ specifies the address of the read port of the FIFO channel to be accessed. In the same way, the processor writes to a FIFO using the write synchronization primitive where the parameter $port$ specifies the address of the write port of this FIFO. The FIFO channels are implemented in the communication memories (CMs); therefore, the addresses of the FIFO ports are located in the processors' address space where the communication memory segment is defined. The memory map of an MPSoC generated by ESPAM contains the values defining the read and the write addresses of each FIFO channel in the system.

The first step in the memory map generation is the mapping of the FIFO channels in the PPN application specification onto the communication memories (CMs) in the multiprocessor platform. This mapping cannot be arbitrary and should obey the mapping rule described at the end of Sect. 30.2. That is, ESPAM maps FIFO channels onto CMs of processors in the following automated way. First, for each process in the application specification ESPAM finds all the channels this process writes to. Then, from the mapping specification ESPAM finds which processor corresponds to the current process and maps the found channels in the processor's local CM. For example, consider the mapping specification shown in Fig. 30.6b which defines only the mapping of the processes of the PPN in Fig. 30.9a to the processors in the platform shown in Fig. 30.9b. Based on this mapping specification, ESPAM maps automatically $FIFO2$, $FIFO3$, and $FIFO5$ onto the

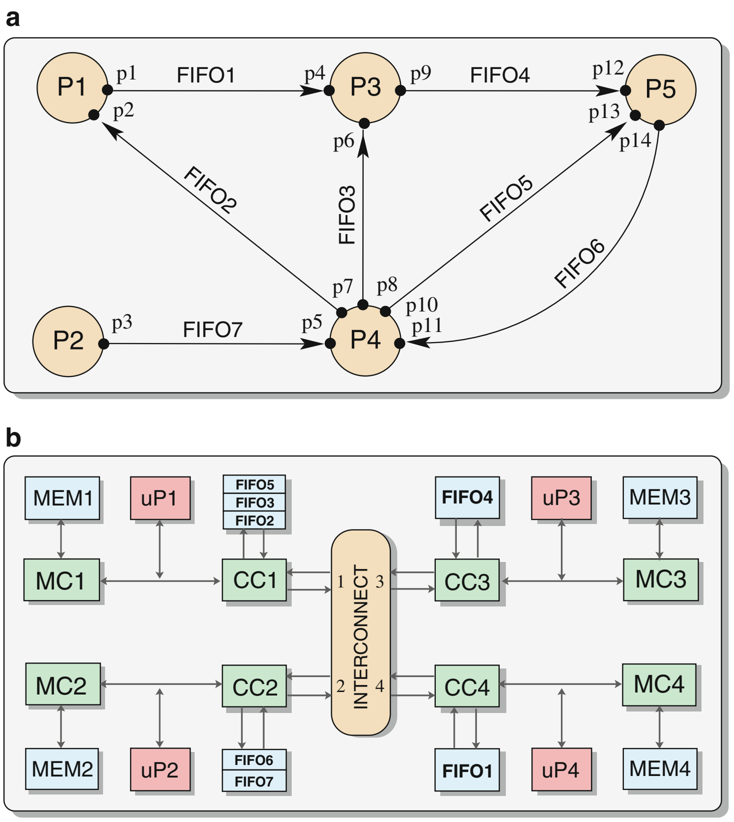


Fig. 30.9 Mapping example. (**a**) Polyhedral process network. (**b**) Example platform

CM of processor $uP1$ because process $P4$ is mapped onto processor $uP1$ and process $P4$ writes to channels $FIFO2$, $FIFO3$, and $FIFO5$. Similarly, $FIFO4$ is mapped onto the CM of processor $uP3$, and $FIFO1$ is mapped onto the CM of $uP4$. Since both processes $P2$ and $P5$ are mapped onto processor $uP2$, ESPAM maps $FIFO6$ and $FIFO7$ onto the CM of this processor.

After the mapping of the channels onto the CMs, ESPAM generates the memory map of the MPSoC, i.e., generates values for the FIFOs' read and write addresses. For the mapping example illustrated in Fig. 30.9b, the generated memory map is shown in Fig. 30.10. Notice that $FIFO1$, $FIFO2$, $FIFO4$, and $FIFO6$ have equal write addresses (see lines 4, 6, 10, and 14). This is not a problem because writing to these FIFOs is done by different processors, and these FIFOs are located in the local CMs of these different processors, i.e., these addresses are local processor write addresses. The same applies for the write addresses of $FIFO3$ and $FIFO7$. However, all processors can read from all FIFOs via a communication component. Therefore, the read addresses have to be unique in the

```
 1 #ifndef  _MEMORYMAP_H_
 2 #define  _MEMORYMAP_H_
 3
 4 #define  p1   0xe0000008 //write addr. FIFO1
 5 #define  p4   0x00040001 //read addr. FIFO1
 6 #define  p7   0xe0000008 //write addr. FIFO2
 7 #define  p2   0x00010001 //read addr. FIFO2
 8 #define  p8   0xe0000010 //write addr. FIFO3
 9 #define  p6   0x00010002 //read addr. FIFO3
10 #define  p9   0xe0000008 //write addr. FIFO4
11 #define  p12  0x00030001 //read addr. FIFO4
12 #define  p10  0xe0000018 //write addr. FIFO5
13 #define  p13  0x00010003 //read addr. FIFO5
14 #define  p14  0xe0000008 //write addr. FIFO6
15 #define  p11  0x00020001 //read addr. FIFO6
16 #define  p3   0xe0000010 //write addr. FIFO7
17 #define  p5   0x00020002 //read addr. FIFO7
18
19 #endif
```

Fig. 30.10 The memory map of the MPSoC platform instance generated by ESPAM

MPSoC memory map and the read addresses have to specify precisely the CM in which a FIFO is located. To accomplish this, a read address of a FIFO has two fields: a communication memory (CM) number and a FIFO number within a CM.

Consider, for example, $FIFO3$ in Fig. 30.9b. It is the second FIFO in the CM of processor $uP1$; thus this FIFO is numbered with 0002 in this CM. Also, the CM of $uP1$ can be accessed for reading through port 1 of the communication component *INTERCONNECT* as shown in Fig. 30.9b; thus this CM is uniquely numbered with 0001. As a consequence, the unique read address of $FIFO3$ is determined to be 0x00010002 – see line 9 in Fig. 30.10, where the first field 0001 is the CM number and the second field 0002 is the FIFO number in this CM. In the same way, ESPAM determines automatically the unique read addresses of the rest of the FIFOs that are listed in Fig. 30.10.

30.6.5 Dedicated IP Core Integration with ESPAM

In Sects. 30.6.3 and 30.6.4 we presented our approach to system-level HW/SW synthesis and code generation for MPSoCs that contain only programmable (ISA) processing components. Based on that, in this section, we present an overview of our approach to augment these MPSoCs with non-programmable dedicated IP cores in a systematic and automated way. Such an approach is needed because, in some cases, an MPSoC that contains only programmable processors may not meet the performance requirements of an application. For better performance and efficiency, in a multiprocessor system, some application tasks may have to be executed by dedicated (customized and optimized) IP cores. Moreover, many companies already provide dedicated customizable IP cores optimized for a particular functionality that aim at saving design time and increasing overall system performance and efficiency. Therefore, we have developed techniques, implemented in ESPAM, for automated generation of an *IP Module* which is a wrapper around a dedicated and predefined IP core. The generated IP Module allows ESPAM to integrate an IP core into an MPSoC in an automated way. The generation of IP Modules is based on the properties of the PPN application model we use in DAEDALUS. Below, we present the basic idea in our IP integration approach. It is followed by a discussion on the type of the IPs supported by ESPAM and the interfaces these IPs have to provide in order to allow automated integration. More details about our integration approach can be found in [36].

30.6.5.1 IP Module: Basic Idea and Structure

As we explained earlier, in the multiprocessor platforms we consider, the processors execute code implementing PPN processes and communicate data between each other through FIFO channels mapped onto communication memories. Using communication controllers, the processors can be connected via a communication component. We follow a similar approach to connect an IP Module to other IP Modules or programmable processors in an MPSoCs. We illustrate our approach with the example depicted in Fig. 30.11. We map the PPN in Fig. 30.2a onto the heterogeneous platform shown in Fig. 30.11a. Assume that process $P1$ is executed by processor $uP1$, $P3$ is executed by $uP2$, and the functionality of process $P2$ is implemented as a dedicated (predefined) IP core embedded in an IP Module. Based on this mapping and the PPN topology, ESPAM automatically maps FIFO channels to communication memories (CMs) following the rule that a processing component

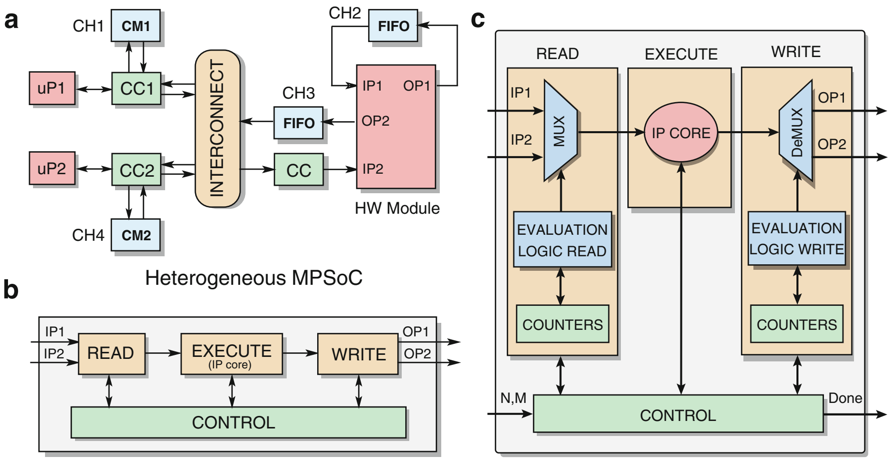


Fig. 30.11 Example of heterogeneous MPSoC generated by ESPAM. (**a**) Heterogeneous MPSoC. (**b**) Top-level view of the IP module. (**c**) IP module structure

only writes to its local CM. For example, process $P1$ is mapped onto processing component $uP1$ and $P1$ writes to FIFO channel $CH1$. Therefore, $CH1$ is mapped onto the local CM of $uP1$ – see Fig. 30.11a. In order to connect a dedicated IP core to other processing components, ESPAM generates an IP Module (IPM) that contains the IP core and a wrapper around it. Such an IPM is then connected to the system using communication controllers (CCs) and communication memories (CMs), i.e., an IPM writes directly to its own local FIFOs and uses CCs (one CC for every input of an IP core) to read data from FIFOs located in CMs of other processors. The IPM that realizes process $P2$ is shown in Fig. 30.11b.

As explained in Sect. 30.3, the processes in a PPN have always the same structure. It reflects the PPN operational semantics, i.e, read-execute-write using blocking read/write synchronization mechanism. Therefore, an IP Module realizing a process of a PPN has the same structure, shown in Fig. 30.11b, consisting of *READ*, *EXECUTE*, and *WRITE* components. A *CONTROL* component is added to capture the process behavior, e.g., the number of process firings, and to synchronize the operation of components *READ*, *EXECUTE*, and *WRITE*. The *EXECUTE* component of an IPM is actually the dedicated IP core to be integrated. It is not generated by ESPAM but it is taken from a library. The other components *READ*, *WRITE*, and *CONTROL* constitute the wrapper around the IP core. The wrapper is generated fully automatically by ESPAM based on the specification of a process to be implemented by the given IPM. Each of the components in an IPM has a particular structure which we illustrate with the example in Fig. 30.11c. Figure 30.2c shows the specification of process $P2$ in the PPN of Fig. 30.2a if $P2$ would be executed on a programmable processor. We use this code to show the relation with the structure of each component in the IP Modules generated by ESPAM, shown in Fig. 30.11c, when $P2$ is realized by an IP Module.

In Fig. 30.2c, the read part of the code is responsible for getting data from proper FIFO channels at each firing of process $P2$. This is done by the code lines 5–8 which behave like a multiplexer, i.e., the internal variable in_0 is initialized with data taken either from port $IP1$ or $IP2$. Therefore, the read part of $P2$ corresponds to the multiplexer MUX in the *READ* component of the IP Module in Fig. 30.11c. Selecting the proper channel at each firing is determined by the *if* conditions at lines 5 and 7. These conditions are realized by the EVALUATION LOGIC READ sub-component in component *READ*. The output of this sub-component controls the MUX sub-component. To evaluate the *if* conditions at each firing, the iterators of the *for* loops at lines 3 and 4 are used. Therefore, these *for* loops are implemented by counters in the IP Module – see the COUNTERS sub-component in Fig. 30.11c.

The write part in Fig. 30.2c is similar to the read part. The only difference is that the write part is responsible for writing the result to proper channels at each firing of $P2$. This is done in code lines 10–13. This behavior is implemented by the demultiplexer DeMUX sub-component in the *WRITE* component in Fig. 30.11c. DeMUX is controlled by the EVALUATION LOGIC WRITE sub-component which implements the *if* conditions at lines 10 and 12. Again, to implement the *for* loops, ESPAM uses a COUNTERS sub-component. Although, the counters correspond to the control part of process $P2$, ESPAM implements them in both the *READ* and

WRITE blocks, i.e., it duplicates the *for*-loops implementation in the IP Module. This allows the operation of components *READ*, *EXECUTE*, and *WRITE* to overlap, i.e., they can operate in pipeline which leads to better performance of the IP Module.

The execute part in Fig. 30.2c represents the main computation in $P2$ encapsulated in the function call at code line 9. The behavior inside the function call is realized by the dedicated IP core depicted in Fig. 30.11c. As explained above, this IP core is not generated by ESPAM but it is taken from a library of predefined IP cores provided by a designer. An IP core can be created by hand or it can be generated automatically from *C* descriptions using high-level synthesis tools like, e.g., Xilinx Vivado [58]. In the IP Module, the output of sub-component MUX is connected to the input of the IP core, and the output of the IP core is connected to the input of sub-component DeMUX. In the example, the IP core has one input and one output. In general, the number of inputs/outputs can be arbitrary. Therefore, every IP core input is connected to one MUX and every IP core output is connected to one DeMUX.

Notice that the loop bounds at lines 3–4 in Fig. 30.2c are parameterized. The *CONTROL* component in Fig. 30.11c allows the parameter values to be set/modified from outside the IP Module at run time or to be fixed at design time. Another function of component *CONTROL* is to synchronize the operation of the IP Module components and to make them to work in pipeline. Also, *CONTROL* implements the blocking read/write synchronization mechanism. Finally, it generates the status of the IP Module, i.e., signal $Done$ indicates that the IP Module has finished an execution.

30.6.5.2 IP Core Types and Interfaces

In this section we describe the type of the IP cores that fit in our IP Module idea and structure discussed above. Also, we define the minimum data and control interfaces these IP cores have to provide in order to allow automated integration in MPSoC platforms generated by ESPAM.

1. In the IP Module, an IP core implements the main computation of a PPN process which in the initial sequential application specification is represented by a function call. Therefore, an IP core has to behave like a function call as well. This means that for each input data, read by the IP Module, the IP core is *executed* and produces output data after an arbitrary delay;
2. In order to guarantee seamless integration within the data-flow of our heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Good examples of such IP cores are the *multimedia cores* at http://www.cast-inc.com/cores/;
3. To synchronize an IP core with the other components in the IP Module, the IP core has to provide `Enable/Valid` control interface signals. The `Enable` signal is a control input to the IP core and is driven by the *CONTROL* component in the IP Module to enable the operation of the IP core when input data is read

from input FIFO channels. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then `Enable` is used to suspend the operation of the IP core. The `Valid` signal is a control output signal from the IP and is monitored by component *CONTROL* in order to ensure that only valid data is written to output FIFO channels connected to the IP Module.

30.7 Summary of Experiments and Results

As a proof of concept, the DAEDALUS methodology/framework and its individual tools (PNGEN, SESAME, and ESPAM) have been tested and evaluated in experiments and case studies considering several streaming applications with different complexity ranging from image processing kernels, e.g., Sobel filter and discrete wavelet transform (DWT), to complete applications, e.g., Motion-JPEG encoder (MJPEG), JPEG2000 codec, JPEG encoder, H.264 decoder, and medical image registration (MIR). For the description of these experiments, case studies, and the obtained results, we refer the reader to the following publications: [36,37] for Sobel and DWT, [34, 36, 37, 51] for MJPEG, [1] for JPEG2000, [38] for JPEG, [46] for H.264, and [13] for MIR. In this section, we summarize very briefly the JPEG encoder case study [38] in order to highlight the improvements, in terms of performance and design productivity, that can be achieved by using DAEDALUS on an industry-relevant application. This case study, which we conducted in a project together with an industrial partner, involves the design of a JPEG-based image compression MPSoC for very high-resolution (in the order of gigapixels) cameras targeting medical appliances. In this project, the DAEDALUS framework was used for design space exploration (DSE) and MPSoC implementation, both at the level of simulations and real MPSoC prototypes, in order to rapidly gain detailed insight on the system performance. Our experience showed that all conducted DSE experiments and the real implementation of 25 MPSoCs (13 of them were heterogeneous MPSoCs) on an FPGA were performed in a short amount of time, 5 days in total, due to the highly automated DAEDALUS design flow. Around 70% of this time was taken by the low-level commercial synthesis and place-and-route FPGA tools. The obtained implementation results showed that the DAEDALUS high-level MPSoC models were capable of accurately predicting the overall system performance, i.e., the performance error was around 5%. By exploiting the data- and task-level parallelism in the JPEG application, DAEDALUS was able to deliver scalable MPSoC solutions in terms of performance and resource utilization. We were able to achieve a performance speedup of up to 20x compared to a single processor system. For example, a performance speedup of 19.7x was achieved on a heterogeneous MPSoC which utilizes 24 parallel cores, i.e., 16 MicroBlaze programmable processor cores and 8 dedicated hardware IP cores. The dedicated hardware IP cores implement the Discrete Cosine Transform (DCT) within the JPEG application. The MPSoC system performance was limited by the available on-chip FPGA memory resources and the available dedicated hardware IP cores in the DAEDALUS RTL library (we had only the dedicated DCT IP core available).

30.8 Conclusions

In this chapter, we have presented our system design methods and techniques that are implemented and integrated in the DAEDALUS design/tool flow for automated system-level synthesis, implementation, and programming of streaming multiprocessor embedded systems on chips. DAEDALUS features automated application parallelization (the PNGEN tool), automated system-level DSE (the SESAME tool), and automated system-level HW/SW synthesis and code generation (the ESPAM tool). This automation significantly reduces the design time starting from a functional specification and going down to complete MPSoC implementation. Many experiments and case studies have been conducted using DAEDALUS, and we could conclude that DAEDALUS helps an MPSoC designer to reduce the design and programming time from several months to only a few days as well as to obtain high quality MPSoCs in terms of performance and resource utilization.

In addition to the well-established methods and techniques, presented in this chapter, DAEDALUS has been extended with new advanced techniques and tools for designing *hard-real-time* embedded streaming MPSoCs. This extended version of DAEDALUS is called DAEDALUS^{RT} [2–5,28,47]. Its extra features are (1) support for multiple applications running simultaneously on an MPSoC; (2) very fast, yet accurate, schedulability analysis to determine the minimum number of processors needed to schedule the applications; and (3) usage of hard-real-time multiprocessor scheduling algorithms providing temporal isolation to schedule the applications.

References

1. Azkarate-askasua M, Stefanov T (2008) JPEG2000 image compression in multi-processor system-on-chip. Tech. rep., CE-TR-2008-05, Delft University of Technology, The Netherlands
2. Bamakhrama M, Stefanov T (2011) Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In: Proceedings of the EMSOFT 2011, pp 195–204
3. Bamakhrama M, Stefanov T (2012) Managing latency in embedded streaming applications under hard-real-time scheduling. In: Proceedings of the CODES+ISSS 2012, pp 83–92
4. Bamakhrama M, Stefanov T (2013) On the hard-real-time scheduling of embedded streaming applications. Des Autom Embed Syst 17(2):221–249
5. Bamakhrama M, Zhai J, Nikolov H, Stefanov T (2012) A methodology for automated design of hard-real-time embedded streaming systems. In: Proceedings of the DATE 2012, pp 941–946
6. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the CODES+ISSS 2003, pp 19–24
7. Clauss P, Fernandez F, Garbervetsky D, Verdoolaege S (2009) Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. IEEE Trans VLSI Syst 17(8):983–996
8. Coffland JE, Pimentel AD (2003) A software framework for efficient system-level performance evaluation of embedded systems. In: Proceedings of the SAC 2003, pp 666–671
9. Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evol Comput 10(3):358–374
10. Erbas C, Pimentel AD (2003) Utilizing synthesis methods in accurate system-level exploration of heterogeneous embedded systems. In: Proceedings of the SiPS 2003, pp 310–315

11. Erbas C, Pimentel AD, Thompson M, Polstra S (2007) A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J Embed Syst 2007(1):1–11
12. Erbas C, Polstra S, Pimentel AD (2003) IDF models for trace transformations: a case study in computational refinement. In: Proceedings of the SAMOS 2003, pp 178–187
13. Farago T, Nikolov H, Klein S, Reiber J, Staring M (2010) Semi-automatic parallelisation for iterative image registration with B-splines. In: International workshop on high-performance medical image computing for image-assisted clinical intervention and decision-making (HP-MICCAI'10)
14. Feautrier P (1988) Parametric integer programming. Oper Res 22(3):243-268
15. Feautrier P (1991) Dataflow analysis of scalar and array references. Int J Parallel Program 20(1):23–53
16. Feautrier P (1996) Automatic parallelization in the polytope model. In: Perrin GR, Darte A (eds) The data parallel programming model. Lecture notes in computer science, vol 1132. Springer, Berlin/Heidelberg, pp 79–103
17. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston
18. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic System-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530
19. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic, Dordrecht
20. Kahn G (1974) The semantics of a simple language for parallel programming. In: Proceedings of the IFIP Congress 74. North-Holland Publishing Co.
21. Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
22. Kienhuis B, Deprettere EF, van der Wolf P, Vissers KA (2002) A methodology to design programmable embedded systems: the Y-chart approach. In: Embedded processor design challenges, LNCS, vol 2268. Springer, pp 18–37
23. Kienhuis B, Rijpkema E, Deprettere E (2000) Compaan: deriving process networks from Matlab for embedded signal processing architectures. In: Proceedings of the CODES 2000, pp 13–17
24. Lee E, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(12):1217–1229
25. Leiden University: The ESPAM tool. http://daedalus.liacs.nl/espam/
26. Leiden University: The PNgen tool. http://daedalus.liacs.nl/pngen/
27. Leiden University and University of Amsterdam: The DAEDALUS System-level Design Framework. http://daedalus.liacs.nl/
28. Liu D, Spasic J, Zhai J, Stefanov T, Chen G (2014) Resource optimization for CSDF-modeled streaming applications with latency constraints. In: Proceedings of the DATE 2014, pp 1–6
29. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the design automation conference (DAC'06), pp 274–279
30. Meijer S, Nikolov H, Stefanov T (2010) Combining process splitting and merging transformations for polyhedral process networks. In: Proceedings of the ESTIMedia 2010, pp 97–106
31. Meijer S, Nikolov H, Stefanov T (2010) Throughput modeling to evaluate process merging transformations in polyhedral process networks. In: Proceedings of the DATE 2010, pp 747–752
32. Mihal A, Keutzer K (2003) Mapping concurrent applications onto architectural platforms. In: Jantsch A, Tenhunen H (eds) Networks on chip. Kluwer Academic Publishers, Boston, pp 39–59
33. Muller HL (1993) Simulating computer architectures. Ph.D. thesis, Department of Computer Science, University of Amsterdam
34. Nikolov H, Stefanov T, Deprettere E (2006) Multi-processor system design with ESPAM. In: Proceedings of the CODES+ISSS 2006, pp 211–216

35. Nikolov H, Stefanov T, Deprettere E (2007) Efficient external memory interface for multi-processor platforms realized on FPGA chips. In: Proceedings of the FPL 2007, pp 580–584
36. Nikolov H, Stefanov T, Deprettere E (2008) Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM. EURASIP J Embed Syst 2008(Article ID 726096)
37. Nikolov H, Stefanov T, Deprettere E (2008) Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(3):542–555
38. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference (DAC'08), pp 574–579
39. Pimentel A, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
40. Pimentel A, Stefanov T, Nikolov H, Thompson M, Polstra S, Deprettere E (2008) Tool integration and interoperability challenges of a system-level design flow: a case study. In: Embedded computer systems: architectures, modeling, and simulation. Lecture notes in computer science, vol 5114. Springer, Berlin/Heidelberg, pp 167–176
41. Pimentel AD, Erbas C (2003) An IDF-based trace transformation method for communication refinement. In: Proceedings of the DAC 2003, pp 402–407
42. Pimentel AD, Polstra S, Terpstra F, van Halderen AW, Coffland JE, Hertzberger LO (2002) Towards efficient design space exploration of heterogeneous embedded media systems. In: Embedded processor design challenges, LNCS, vol 2268. Springer, pp 57–73
43. Pimentel AD, Thompson M, Polstra S, Erbas C (2006) On the calibration of abstract performance models for system-level design space exploration. In: Proceedings of the SAMOS'06, pp 71–77
44. Piscitelli R, Pimentel AD (2011) A high-level power model for MPSoC on FPGA. In: Proceedings of the IPDPS – IEEE RAW workshop 2011, pp 128–135
45. Piscitelli R, Pimentel AD (2012) A signature-based power model for MPSoC on FPGA. VLSI Design 2012:1–13
46. Rao A, Nandy SK, Nikolov H, Deprettere EF (2011) USHA: unified software and hardware architecture for video decoding. In: Proceedings of the SASP 2011, pp 30–37
47. Spasic J, Liu D, Cannella E, Stefanov T (2015) Improved hard real-time scheduling of CSDF-modeled streaming applications. In: Proceedings of the CODES+ISSS 2015, pp 65–74
48. Stefanov T, Deprettere E (2003) Deriving process networks from weakly dynamic applications in system-level design. In: Proceedings of the CODES+ISSS 2003, pp 90–96
49. Stefanov T, Kienhuis B, Deprettere E (2002) Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proceedings of the CODES 2002, pp 7–12
50. van Stralen P, Pimentel AD (2012) A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs. In: Proceedings of the CODES+ISSS 2012, pp 393–402
51. Thompson M, Nikolov H, Stefanov T, Pimentel A, Erbas C, Polstra S, Deprettere E (2007) A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: Proceedings of the CODES+ISSS 2007, pp 9–14
52. Turjan A, Kienhuis B, Deprettere E (2004) Translating affine nested-loop programs to process networks. In: Proceedings of the CASES 2004, pp 220–229
53. University of Amsterdam: The SESAME tool. https://csa.science.uva.nl/download/
54. Verdoolaege S (2010) Polyhedral process networks. In: Handbook of signal processing systems. Springer US, pp 931–965
55. Verdoolaege S, Bruynooghe M, Janssens G, Catthoor F (2003) Multi-dimensional incremental loop fusion for data locality. In: Proceedings of the ASAP 2003, pp 17–27
56. Verdoolaege S, Nikolov H, Stefanov T (2007) pn: a tool for improved derivation of process networks. EURASIP J Embed Syst 2007(1):1–13

57. Verdoolaege S, Seghir R, Beyls K, Loechner V, Bruynooghe M (2004) Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In: Proceedings of the CASES 2004, pp 248–258
58. Xilinx, Inc. Vivado high-level synthesis from Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html
59. Zhai J, Nikolov H, Stefanov T (2013) Mapping of streaming applications considering alternative application specifications. ACM Trans Embed Comput Syst 12(1s):34:1–34:21