Chapter 2

Daedalus: design flow

2.1 Introduction

As mentioned in the previous chapter, the complexity of modern embedded systems, which are increasingly based on heterogeneous MultiProcessor-SoC (MP-SoC) 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. Key enablers to this end are, for example, the use of architectural platforms to facilitate re-use of IP components and the notion of high-level system modeling and simulation [53]. 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 MP-SoC 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 MP-SoC architecture [66]. Subsequently, applications need to be partitioned into HW and SW parts since MP-SoC architectures often are heterogeneous in nature. To this end, MP-SoC platform architectures need to be modeled and simulated 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, which involves the synthesis of its architectural components 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 remains a large gap (the so-called implementation gap [69]) between the deployed system-level models and actual implementations of the system under study. Currently, there exist no mature methodologies, techniques, and tools to effectively and efficiently convert system-level system specifications to RTL specifications.

Daedalus’ main objective is to bridge the aforementioned implementation gap for the
design of multimedia MP-SoCs. It does so by providing an integrated and highly-automated environment for system-level architectural exploration, system-level synthesis, programming and prototyping. In this chapter we will show how the different components fit together as the pieces of a puzzle, resulting in a system-level design environment that addresses the entire design trajectory with an unparalleled degree of automation.

The next section provides a birds-eye overview of Daedalus, after which the three subsequent sections present the three core tools that constitute Daedalus in more detail. More specifically, Section 2.3 explains how multimedia applications are automatically decomposed in parallel specifications. Section 2.4 describes how – given the parallel application(s) – promising candidate architectures can be found using our system-level modeling, simulation and exploration methodology and toolset. In Section 2.5, we explain how selected candidate architectures can be automatically and rapidly synthesized, programmed and prototyped. Section 2.6 describes in some more detail how the different components of the tool-flow have been linked together. Finally, Section 2.7 discusses related work. As the main focus of this thesis concerns design space exploration, the next chapter will take a closer look at the Sesame simulation and modeling environment, which we already introduced shortly in this chapter (Section 2.4 In Chapter 8 of this thesis a case study demonstrates the entire Daedalus design flow in action.

2.2 The Daedalus framework

In Figure 2.1, the design flow of the Daedalus framework is depicted. As mentioned before, Daedalus provides a single environment for rapid system-level architectural exploration, high-level synthesis, programming and prototyping of multimedia MP-SoC architectures. Here, a key assumption is that the MP-SoCs are constructed from a library of pre-determined and pre-verified IP components. These components include a variety of programmable and dedicated processors, memories and interconnects, thereby allowing the implementation of a wide range of MP-SoC platforms. The remainder of this section provides a high-level overview of Daedalus, after which the subsequent sections zoom in on its core components and how they interact with the rest of the design flow.

Starting from a sequential application specification in C or C++, the KPNgen tool [116] allows for automatically converting the sequential application into a parallel Kahn Process Network (KPN) [51] specification. Here, the sequential input specifications are restricted to so-called static affine nested loop programs, which is an important class of programs in, e.g., the scientific and multimedia application domains. By means of automated source-level transformations [94], KPNgen is also capable of producing different input-output equivalent KPNs, in which for example the degree of parallelism can be varied. Such transformations enable application-level design space exploration.

The generated or handcrafted KPNs (the latter in the case that, e.g., the input specification did not entirely meet the requirements of the KPNgen tool) can subsequently be used by our Sesame modeling and simulation environment [79] to perform system-level architectural DSE. To this end, Sesame uses (high-level) architecture model components from the
2.2. THE DAEDALUS FRAMEWORK

IP component library. Sesame allows for quickly evaluating the performance of different application to architecture mappings, HW/SW partitionings, and target platform architectures. Such DSE should result in a number of promising candidate system designs, of which their specifications (system-level platform description, application-architecture mapping description, and application description) act as input to the ESPAM tool [74]. This tool uses these system-level input specifications, together with RTL versions of the components from the IP library, to automatically generate synthesizable VHDL that implements the candidate MP-SoC platform architecture. In addition, it also generates the C/C++ code for those application processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can be readily mapped onto an FPGA for prototyping. Such prototyping also allows for calibrating and validating Sesame’s system-level models, and as a consequence, improving the trustworthiness of these models.

Ultimately, Daedalus aims at traversing an entire design flow – going from a sequential application to a working MP-SoC prototype in FPGA technology with the application mapped onto it – in a matter of hours. Evidently, this would offer great potentials for quickly experimenting with different MP-SoC architectures and exploring design options during the early stages of design.

In the following sections we describe each of the main components in Daedalus: appli-
2.3 Parallelizing applications

Today, traditional imperative languages like C or C++ are still dominant with respect to implementing applications for SoC-based architectures. It is, however, difficult to map these imperative implementations, with typically a sequential model of computation, onto MP-SoC architectures that allow for exploiting task-level parallelism in applications. In contrast, models of computation that inherently express task-level parallelism in applications and make communications explicit, such as CSP [43] and Process Networks [51], allow for easier mapping onto MP-SoC architectures. However, specifying applications using these models of computation usually requires more implementation effort in comparison to sequential imperative solutions.

In Daedalus, we start from a sequential imperative application specification (C/C++) which is then automatically converted into a Kahn Process Network (KPN) [51] using the KPNgen tool [116]. This conversion is fast and correct by construction. In the KPN model of computation, parallel processes communicate with each other via unbounded FIFO channels. Reading from channels is done in a blocking manner, while writing to channels is non-blocking. We use KPNs for application specifications because this model of computation nicely fits the targeted media-processing application domain and is deterministic. The latter implies that the same application input always results in the same application output, irrespective of the scheduling of the KPN processes. This provides complete scheduling freedom when, as will be discussed later on, mapping KPN processes onto MP-SoC architecture models for quantitative performance analysis and design space exploration.

As mentioned before, KPNgen’s input applications need to be specified as so-called static affine nested loop programs to allow for automatic parallelization of applications. As a first step, KPNgen can apply a variety of source-level transformations to these specifications in order to, for example, increase or decrease the amount of parallelism in the final KPN [94]. Subsequently, the C/C++ code is transformed into single assignment code (SAC), which resembles the dependence graph (DG) of the original nested loop program. Hereafter, the SAC is converted to a Polyhedral Reduced Dependency Graph (PRDG) data structure, being a compact mathematical representation of a DG in terms of polyhedra. Finally, a PRDG is converted into a KPN by associating a KPN process with each node in the PRDG. The parallel KPN processes communicate with each other according to the data dependencies given in the DG.

In Figure 2.2, a Kahn Process Network example is given in which three processes (A, B and C) are connected using three channels (CH1-3). Figure 2.2(a) shows the XML description of Kahn process B as generated by KPNgen. The XML describes both the topology of the KPN (i.e., how the processes are connected together, see e.g., lines 20-25) as well as the communications and computations performed by processes. In our example, process B executes a function called compute (line 8). The function has one input argument (line 9) and one output argument (line 10). The relation between the function arguments and the
2.3. PARALLELIZING APPLICATIONS

From the XML specification, Daedalus allows for automatically generating the C/C++ code implementing the behavior of each KPN process. This is done by the ESPAM tool, which will be discussed later on. Figure 2.2(b) shows, for example, the generated C code for process B (some variable declarations have been omitted). The code contains the main behavior of a process, together with the read/write communication primitives. In accordance with the XML specification in Figure 2.2(a), the function compute – which is derived from the original sequential application specification – is part of a loop that iterates $2 \times N - 2$ times. For synthesis purposes, Daedalus also allows for generating the code for the read and write communication primitives, as shown in Figure 2.2(b). Currently, these primitives are implemented using polling and memory-mapped I/O. Note that the implementation of the write primitive is blocking since at implementation level FIFO channels are bounded in size.

communication ports of the process is given in lines 3 and 6. The function has to be executed $2 \times N - 2$ times as specified by the polytope in lines 12-13. The value of $N$ is between 3 and 384 (lines 14-15).

![Figure 2.2: A Kahn Process Network example.](image-url)
CHAPTER 2. DAEDALUS: DESIGN FLOW

2.4 Design Space Exploration

Given a (set of) KPN application specification(s) – as for example generated by KPNgen or devised by hand – and the components in Daedalus’ IP library, the Sesame system-level simulation framework [79] addresses the problem of finding a suitable and efficient target MP-SoC platform architecture. Figure 2.3 illustrates Sesame’s layered infrastructure for the case in which a Motion-JPEG application is studied with a crossbar-based distributed-memory MP-SoC as target architecture. Sesame deploys separate application and architecture models, where an application model describes the functional behavior of an application and an architecture model defines architecture resources and captures their performance constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for evaluation of the system performance of a particular application, mapping, and underlying architecture. Essential in this methodology is that an application model is independent from architectural specifics and assumptions on hardware/software partitioning. As a result, 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 or modeling the same architecture design at various levels of abstraction.

For application modeling, the computational and communication behavior of the KPN application specifications are captured using application event traces. The computation and communication events in these traces typically are coarse grained, such as Execute(DCT) or Read(channel_id, pixel-block). To generate the application events, the C/C++ code of each Kahn process is instrumented with annotations that describe the application’s computational actions. In addition, Sesame provides read and write communication primitives that generate communication events as a side-effect. So, by executing the KPN model, each process generates its own trace of application events, representing the workload that is imposed on the underlying MP-SoC architecture model.

An architecture model simulates the performance consequences of the computation and
communication events generated by an application model. To this end, each component in
the architecture model is parameterized with performance parameters specifying the laten-
cies of computation events like *Execute(DCT)*, communication transactions, and memory
accesses. This approach allows to quickly assess, e.g., different HW/SW partitionings by
simply experimenting with the latency parameters of processing components in the archi-
tecture model: a low computational latency refers to a HW implementation while a high latency
mimics a SW solution.

To bind application tasks to resources in the architecture model, Sesame provides an in-
termediate *mapping layer*. It controls the mapping of Kahn processes (i.e. their event traces)
onto architecture model components by dispatching application events to the correct archi-
tecture model component. The mapping also includes the mapping of Kahn channels onto
communication resources in the architecture model. The mapping layer has two additional
purposes. First, the event dispatch mechanism in the mapping layer provides a variety of
static and dynamic policies to schedule application tasks (i.e., their event traces) that are
mapped onto shared architecture model components. Second, the mapping layer is also ca-
pable of dynamically transforming application events into (lower-level) architecture events
in order to facilitate flexible refinement of architecture models [79].

The output of system simulations in Sesame provides the designer with performance
estimates of the system(s) under study together with statistical information such as utiliza-
tion of architectural components (idle/busy times), the contention in a system (e.g., network
contention), profiling information (time spent in different executions), critical path analy-
sis, and average bandwidth between architecture components. Such results allow for early
evaluation of different design choices, identifying trends in the systems’ behavior, and can
help in revealing performance bottlenecks early in the design cycle. Here, the exploration
process is also facilitated by the fact that system configurations (bindings, scheduling and
arbitration policies, performance parameters, and so on) are specified using XML descrip-
tions. Hence, different system configurations can be rapidly simulated without remodeling
and/or recompilation.

As a result of the design space exploration with Sesame, a small set of promising MP-
SoC platform instances can be selected for automatic synthesis (see next section). Each
selected platform instance is specified using two XML files. One describing the architectural
platform at the system level, i.e. which IP components are used in the platform and how
they are interconnected. And the other describing how application tasks are mapped onto
the platform components.

### 2.5 System-level Synthesis

The system-level specifications that result from DSE – describing (the structure of) the ap-
lication and platform architecture as well as the mapping of the former onto the latter – are
given as input to the ESPAM tool for system-level synthesis [74]. To guarantee correctness-
by-construction, ESPAM first runs a consistency check on the provided platform instance.
This includes finding impossible and/or meaningless connections between system-level plat-
form components as well as parameter values that are out of range. Subsequently, ESPAM refines the abstract platform model to a parameterized RTL model which is ready for an implementation on a target physical platform. The refined system components are instantiated by setting their parameters based on the target physical platform features. Finally, ESPAM generates program (C/C++) code for each programmable processor in the multiprocessor platform in accordance with the application and mapping specifications. To this end, it uses the XML specifications generated by KPNgen. In addition, ESPAM also provides the support for scheduling the code in the case multiple application processes are mapped onto a single processor in the platform. Currently, this code scheduling is performed statically.

The output of ESPAM, namely an RTL specification of the MP-SoC platform, is a model that can adequately abstract and exploit the key features of a target physical platform at the register transfer level. It consists of four parts (as shown in Figure 2.1): 1) a platform topology description defining in greater detail the structure of the multiprocessor platform; 2) hardware descriptions of IP cores containing predefined and custom IP cores used in 1). These IP cores, which are selected from Daedalus’ IP component library, include programmable as well as dedicated processors, various memory components (FIFO buffers, random access memory, etc.), and different interconnects (point-to-point links, shared bus with various arbitration mechanisms, and a crossbar switch). For programmable processors, ESPAM currently uses PowerPCs and Microblazes since it targets the Xilinx VirtexII-Pro family of FPGA technology for prototyping the synthesized MP-SoCs. ESPAM also automatically generates custom IP cores needed as a glue/interface logic between components in the platform; 3) the program code for processors — as mentioned before, to execute the software parts of the application on the synthesized multiprocessor platform, and 4) Auxiliary information containing files which give tight control on the overall specifications, such as defining precise timing requirements and prioritizing signal constraints.

With the above descriptions, a commercial synthesizer can convert an RTL specification to a gate-level specification, thereby generating the target platform gate-level netlist (see the bottom part of Figure 2.1). At this moment, ESPAM facilitates automated MP-SoC synthesis and programming using the Xilinx VirtexII-Pro family of FPGAs and therefore uses the Xilinx Platform Studio (XPS) tool as a back-end to generate the final bit-stream file that configures the FPGA. However, our framework is general and flexible enough to be targeted to other physical platform technologies as well.

2.6 The Daedalus Design-flow Infrastructure

As discussed in the previously, the heart of Daedalus consists of the three core tools KPNgen, Sesame and ESPAM. In addition, Daedalus also features several supporting tools to improve the user-friendliness and deployability of the framework. This section provides a brief overview of the supporting infrastructure.

In Daedalus, most design information (e.g., structural descriptions of the application, architecture, and the mapping of the former onto the latter) as well as experimental results are described using XML-based descriptions. Daedalus therefore contains the Oracle Berkeley
2.6. **THE DAEDALUS DESIGN-FLOW INFRASTRUCTURE**

---

DB XML relational database management system (RDBMS) to store all information (models, parameters and results) related to designs and experiments. This RDBMS, together with its GUI, provide the designer with a powerful tool to e.g., explore and visualize the large amounts of data generated by Daedalus design space exploration. Moreover, it guarantees the reproducibility of experiments at all times.

The vision behind the Daedalus software infrastructure is that it should be open for integration of new tools as well as that it should allow for customization of the design flow. Therefore, the design flow (or tool flow) in Daedalus is composable and constructed from 'design-flow blocks'. These design-flow blocks, which are illustrated as the dashed boxes in Figure 2.4, are the tools that take part in the design flow together with their input- and output descriptions. The latter descriptions, illustrated by the grey boxes in Figure 2.4, provide information about what input/output data a tool consumes/produces and from/to where it reads/writes this data. This allows us to describe a design flow as a simple composition of the design-flow blocks, specified in the workflow description. For example, Figure 2.4 shows a design flow which includes a visualization block to visualize Sesame’s DSE results and which stores both the DSE and ESPAM’s prototyping results in the RDBMS (using the so-called 'XML saver' tool). Evidently, this composability of the design flow allows for easily adding new design steps to a design flow, or to customize design flows for specific design domains.

Control and monitoring software utilities have been developed to facilitate the process of setting up and executing experiments on the FPGA-based prototypes of MP-SoCs generated by Daedalus. Such utilities are necessary and very useful for: (i) conducting an effective and efficient design space exploration at implementation level on a narrow design space defined by Sesame; (ii) measuring real performance and cost numbers used for calibration of the Daedalus’ high-level architecture models [80]; (iii) preparing real HW/SW demonstrators. The control and monitoring utilities include a configuration manager, an execution control panel, and an on-line monitoring console, all supported by a GUI which allows users unfamiliar with the FPGA prototyping board to perform experiments with the MP-SoCs.

---

Figure 2.4: Daedalus’ customizable work flow.
2.7 Related Work

Systematic and automated application-to-architecture mapping has been widely studied in the research community. The closest to our work is the Koski MP-SoC design flow [52] and the SystemC-based design methodology presented in [40]. Koski provides a single infrastructure for modeling of applications, automatic architectural design space exploration, and automatic system-level synthesis, programming, and prototyping of selected MP-SoCs. The methodology in [40] supports automated design space exploration, performance evaluation, and automatic platform based system generation. But unlike Daedalus, [52] and [40] do not allow for automated parallelization of applications, nor design space exploration at application level. Both [52] and [40] require applications to be specified by hand in UML and SystemC, respectively.

Other examples of related work can be found in [95, 64, 20, 33]. However, these efforts are limited to processor-coprocessor architectures [95], only provide a limited degree of automation [64, 20], or do not provide an automated step towards RTL [33].

Companies such as Xilinx and Altera provide design tool chains attempting to generate efficient implementations starting from descriptions higher than (but still related to) the register transfer level of abstraction. The required input specifications are still so detailed that designing a single processor system is still error-prone and time consuming, let alone designing alternative multiprocessor systems. In contrast, Daedalus raises the design to an even higher level of abstraction allowing the exploration, design, and programming of multiprocessor systems in a short amount of time. For a more detailed discussion of related work and a classification of system-level design and synthesis tools we refer to [34].

2.8 Conclusion

In this chapter we presented the Daedalus framework that tries to bridge the so-called implementation gap between system-level platform specifications and the actual physical implementations of these platforms. To this end, Daedalus focuses on the design of multimedia MP-SoC platforms. As such, it provides an integrated and highly-automated environment for system-level architectural exploration, system-level synthesis, programming and prototyping. Such a framework offers remarkable potentials for quickly experimenting with different MP-SoC architectures and exploring system-level design options during the very early stages of design. In Chapter 8 we illustrate Daedalus’ design steps and demonstrate its efficiency using a case study with a Motion-JPEG encoder application.