Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Entwurfsautomatisierung

Graph-Grammar-Based IP Integration (GRIP) Tool for Efficient IP Reuse in Software-Defined SoCs

Munish Jassi

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Georg Sigl
Prüfer der Dissertation:	1. Prof. Dr.-Ing. Ulf Schlichtmann
	2. apl. Prof. Dr.-Ing. Walter Stechele

Die Dissertation wurde am 27.11.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 28.01.2019 angenommen.

Abstract

In modern system-on-chip (SoC) designs, IP reuse is considered a driving force to increase productivity. To support various designs, a large number of Intellectual Property (IP) hardware blocks have been developed. The integration of those IPs into an SoC may require significant effort - up to days or weeks depending on experience and complexity. This work presents a novel approach to significantly reduce the design effort to bring-up a working SoC design by automatic IP integration as part of a library-based Software-defined SoC flow. In detail, the IP supplier prepares a HW-accelerated software library (HASL) for the SoC architect, who wants to use the IP in an SoC design. As a key point of the presented approach, IP integration knowledge is encoded in the library as a set of integration rules. These rules are defined in the machine-readable standardized IP-XACT format by the IP supplier, who has a good knowledge of the IP's hardware details. The library preparation step on the IP supplier's side is also partly automated in the proposed flow, including generation of configurable HW drivers, schedulers, and the software library functions. For the SoC architect, the graph-grammar-based IP-integration (GRIP) tool is developed. The software application is developed using the functions supplied in the HASL. According to the calls to the HASL functions, the GRIP tool automatically integrates IP blocks using the rule information supplied with the library and runs a full Design Space Exploration. For this, the SoC architecture and rules are transformed into the graph domain to apply graph rewriting methods. The GRIP tool is model-driven and based on the Eclipse Modeling Framework. With code generation techniques, SoC candidate architectures can be transformed to hardware descriptions for the target platform. The HW/SW interfaces between SW library functions and IP blocks can be automatically generated for bare-metal or Linux-based applications.

The approach is demonstrated with two case studies on the Xilinx Zynq-based ZedBoard evaluation board using a HASL for computer vision. It can yield 10x - 150x performance improvement for the bare-metal application versions and 4x - 7x performance improvement for the Linux-based application versions, when executed on an optimized HW-accelerated SoC architecture compared to a non HW-accelerated SoC. The effort for IP integration is comparable to using a software library, hence, providing a significant advantage over a manual IP integration.

Acknowledgements

I would like to thank my supervising professor, Prof. Ulf Schlichtmann, for giving me an opportunity to research on the topic of my interest. His guidance and support during my research work was invaluable, without which this research would not have been possible. I would like to thank my advisor, Dr. Ing.- Daniel Müller-Gritschneder, for all his technical feedbacks and guiding the research project. Daniel has been the first go-to person for any deep-dive discussions, and he has provided valuable inputs on numerous topics. The mentorship from both Prof. Schlichtmann and Daniel went beyond the research to other topics including project management, teaching, and supervision, which I am also carrying further into my next career steps.

It was an honour to collaborate with researchers from Technical University of Munich (TUM), as well as beyond TUM. I would like to thank my colleague Yong Hu for working together on graph grammar algorithms. Thanks to Alessandro Bernardini for his insightful ideas on defining mathematical completeness of algorithms. Thanks to Prof. Walter Stechele for guidance on the optical flow use-case study. More importantly, I owe much credits to all students who helped me on various sub projects including Jian Lyu, Umair Razzaq, and Benjamin Bordes.

I would like to thank my parents - Simro Devi and Mohan Lal - who have always supported and encouraged me to pursue my interests. I would like to thank my friends – Dr. Dinesh Kumar, Vlastimil Zwiefelhofer, Navjot Singh, Inderpal Singh, Vivek Kumar, Shushanik Karapatyan, Andreas Herrmann, Chunfeng Liu, Ravi Pujara, Simona Gavrilova, Irfan Karabegovic and Matthias Fasching – who have not only helped me to keep motivated for my research, but also made my stay at TUM and Munich enjoyable.

This research project is sponsored by the German Research Foundation (DFG). I am grateful to Deutsche Forschungsgemeinschaft (DFG) for contributing to fund this research. This research is partly funded by the DFG under the research grant SCHL 347/3-1.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Introduction and Motivation

Previous decades of research and technology advancements in semiconductors have enabled us to discover unforeseeable applications of electronic devices. The benefits of electronic devices for us are ubiquitous, and are relevant for the production and organization of knowledge, social cooperation, productivity, safety and health. These devices automate our day-to-day tasks, maintain health, enable communication and transportation, distribute information and provide entertainment. In all these tasks, the electronic devices interact with us and our environment to collect relevant data, and further transmit and process the data for desired tasks. The desired tasks are realized by a collection of functions, which are implemented either by hardware (HW) components or by software (SW) applications executed on programmable hardware processing units.

The advancements of innovative software applications, supporting our improving quality of life, are demanding higher computation capabilities from electronic devices. In the previous decades, we have made significant technical progresses and breakthroughs to implement increasingly complex tasks on electronic devices. The technology breakthroughs have lead us to shrink HW elements, including analog and digital HW components, HW processing units etc.. This progressive HW shrinking has helped to combine multiple HW elements in a single chip, called system-on-chip (SoC). The SoCs have now become fundamental building blocks of any electronic devices. By integrate increasingly complex HW elements into an SoC has significantly improved computation capabilities. These computation improvements are primarily because of the availability of increasing number of computational resources on a same chip, and by avoiding slow off-chip data communications. A typical electronic device contains multiple SoCs to perform required digital or analog processing tasks.

An SoC contains multiple computational resources connected via various data communication channels. The computational resources on an SoC are the HW modules that process, store or transmit data. The communication channels on an SoC are the wire connections that transmits data and signals among HW modules. In order to support increasing complexities of SoCs, it requires increasing design effort from the SoC designers. Studies have shown that SoC designers are unable to cope with the increasing SoC design complexities. Fig. 1.1 shows the increasing gap between the SoC designs complexity and the designers' productivity (SEMATECH 2011). The SoC design complexity is still following Moore's law of 58% annual increase, while the SoC designers are seeing

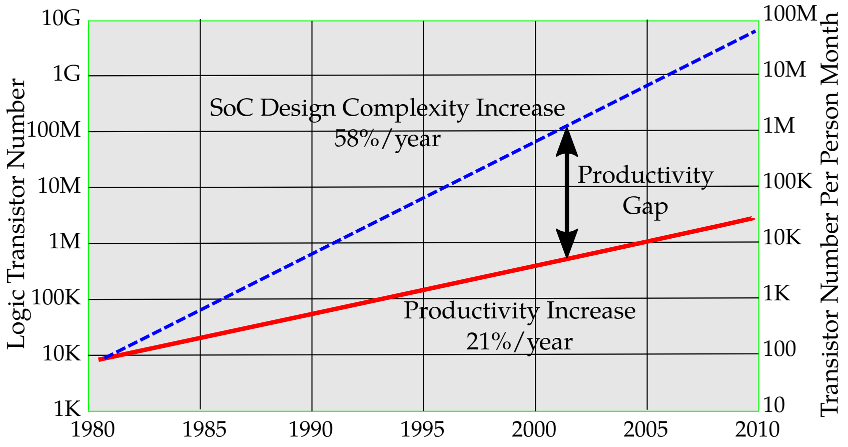


Figure 1.1.: Increasing gap between the SoC complexity and designer's productivity

productivity increase of 21% annually with the improving design methodologies. This require novel solutions to keep up with the increasing SoC complexities.

Many industrial SoC design methodologies use divide and conquer approaches as a strategy to handle the complexity of state-of-the-art SoCs. This has led to the concepts of intellectual property (IP) based design flow and IP reuse for SoC implementation. In the IP based design flow, some computationally heavy SW tasks of an application are implemented on dedicated HW IP components. The dedicated SW tasks may include handling communication protocols (e.g. UART, Ethernet, PCIe), dedicated data processing units (e.g. data encryption), or domain-specific functions (e.g. Fourier transforms, signal and image processing filters). These HW IPs are independently optimized and implemented, and later integrated into an SoC design. The HW IPs such implemented can be reused in multiple SoC designs. The international technology roadmap for semiconductors (ITRS 2011) includes a prediction for the amount of HW IP reuse in future system-on-chip (SoC) designs. It states that IP reuse in SoCs will grow from around 70% in 2016 to more than 90% by 2020. Thus, IP-based SoC design methodologies have gained attention in industry as well as in academia. The IP reuse promotes using IP components available from either previous design projects or third-party IP suppliers. In IP-based flows, there are two primary actors: the IP supplier and the IP user, in the following referred as the SoC architect. The challenges in IP-based flows are mainly related to an increasing knowledge gap among these actors. The challenges are related to both the SoC HW implementation, and the associated HW drivers and SW application implementation to bring-up the associated HW-SW system.

In IP-based design, an IP supplier owns a dedicated HW IP or a domain-specific IP library containing dedicated HW IPs for domain-specific functions (e.g. Computer vision domain). The IP supplier progressively optimizes the HW IPs, and transfers them to the SoC architect either as hard IPs or soft IPs. These HW IPs are associated with HW drivers, which are required by SW applications to access the HW IPs. An objective for the IP supplier is to prepare an IP library containing sufficient knowledge on efficiently integrating the provided IPs into an SoC, and prepare required HW drivers. A hindrance for the IP supplier is his limited knowledge about the scope of SoCs in which the IPs will be utilized. In the existing design methodologies, the IP integration knowledge is provided as documentations in the IP library. While utilizing the IPs available from IP suppliers, the challenges for an SoC architect are efficiently integrating IPs to implement a synthesizable SoC, implementing and configuring HW drivers, adapting SW application to utilize available HW resources, and for the systems with operating systems (OS) - preparing kernel drivers and board support packages (BSP). An inefficient IP integration, or IP handling using the HW drivers can incur significant performance overhead because of data communication and HW-access latency. This process of IP integration using the available documentations can be an error-prone and laborious task.

In order to find optimal SoC candidates for targeted SW applications, the SoC architect must perform design space exploration (DSE) to guarantee desired SW performances. The SoC DSE utilizing the available IP library must also obey to SoC design constraints of HW resources area, power, and latencies. With multiple available HW IPs in an IP library, there are primarily two challenges for the SoC architect, first, find an optimal set of HW IPs to accelerate target SW applications, and second, integrate the HW IPs to the SoC design in efficient architectural configurations. Each set of SoC structural modifications associated with the DSE iteration requires re-implementation of HW drivers stack and adaptation of the SW application to utilize additional HW IPs. The number of candidate SoCs during DSE grows exponentially with increasing HW IPs. The challenges for an SoC architect during DSE are to constrain the SoC design space to quickly find the optimal candidate SoCs and evaluate their performances.

The SoCs are primarily realized as application-specific integrated circuits (ASICs). Many of the ASIC design flows utilize field programmable gate array (FPGA) platforms for SoC prototyping, performance evaluations, and eventual design space explorations. With recent advances in FPGA technology, FPGAs are no more only a cluster of programmable logic. Some recent FPGA chipsets, like Xilinx Zynq7000 series (ZedBoard 2012), provide powerful microprocessors with other essential interfacing IP hard-cores and a dedicated programmable fabric interfaced via multiple high-speed buses. These FPGAs are self-sufficient to prototype a complete or a sub-system of an SoC. In recent years, these FPGAs are been used for prototyping as well as implementing application-specific SoCs. For some application domains off-the-shelf ASICs are not able to meet the desired performance requirements. SoC architects are opting for FPGAs as an alternative solution to ASICs for these highly customized SoCs for targeted applications.

The design flexibility offered by these FPGAs has led to the concept of software-defined SoCs. In this approach, first, the target software applications are executed on an SoC with bare-minimum HW computational resources. Progressively, some of the SW tasks are transferred to additional dedicated HW computing resources (hardware accelerators). For DSE, the SoC architect must explore feasible SoC architecture candidates using an IP

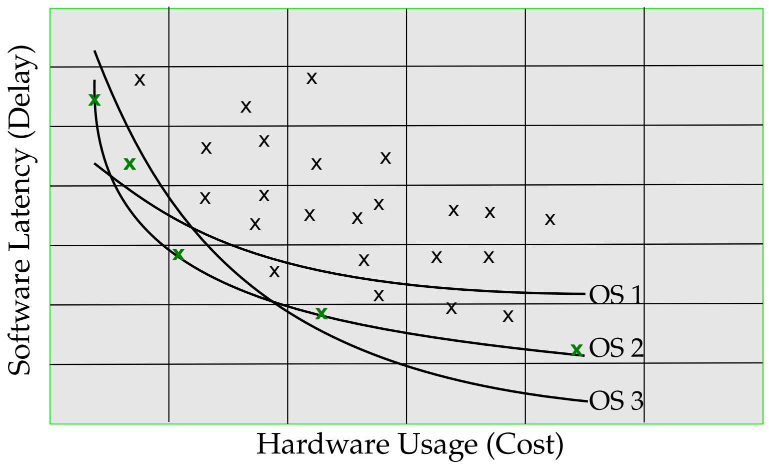


Figure 1.2.: SW performance vs. HW cost trade offs under different operating systems

library for a targeted SW application. Fig. 1.2 shows a representative distribution of the SW performance vs. HW cost trade offs for SoC architectures under different operating systems (OSs). In the figure, the 'x's represent candidate SoCs obtained by integrating different combination of HW IPs (in the figure, the 'x's are targeted on OS2). The candidate SoCs that are closest to the origin provide the optimal performance-cost trade offs (Green colored 'x's). These optimal SoCs form the Pareto optimal front under OS 2. The set of SoCs on the Pareto optimal front changes with different operating systems. This is because each OS is associated with different data handling and communication. So, SoC DSE needs to be performed separately for each targeted OS. Evaluating the candidacy of each SoC architecture is very expensive. It requires the HW synthesis, SW application adaptation, and finally the system bring-up and evaluations. So, one of the objectives during the SoC DSE is also to eliminate the non-optimal SoC candidates, while still keeping the Pareto optimal SoC candidates.

This work tries to tackle the challenges of SW-defined SoCs approach related to the IP packaging, IP-library preparation, IP Integration and the SoC design space exploration. The next sections will provide the problem statements for this thesis and summarize the contributions of this work.

1.2. Problem Statement

The core problem addressed in this work is to bridge the knowledge gap among the IP suppliers and the SoC architects, while strengthening IP-based SoC design methodologies and IP reuse. This is to further help reducing the productivity gap between SoC designers' productivity and demands for increasing SoC complexities. In the IP-based SoC design flow, one challenge for an IP supplier is to efficiently encode the IP integration knowledge. The key challenges for an SoC architect are to efficiently integrate IPs, synthesize SoC on HW platforms, implement HW drivers, utilize HW drivers in the SW application for improving performances, and perform SoC design space exploration.

Current SW-defined SoC design flows are limited to semi-automated approaches using high level synthesis for HW IP implementation. This significantly limits the usage of available third-party HW IPs or IP libraries, restricts the SoC to limited architectural configurations, and increases the effort to perform the SoC design space exploration. Existing problems in the SW-defined SoCs include inefficient IP exchange and IP library preparation, and lack of automated IP integration and SoC design space exploration targeted on a domain-specific hardware IP library. This work tries to handle these problems in the SW-defined SoC design while widening its scope to include third-party domain-specific IP libraries.

The SW-defined SoC design and optimization is targeted to enable SoC system bring-up on an FPGA platform. The HW-SW system prototyping on an FPGA gives better performance correlation to the final SoC. However, SoC prototyping and DSE on an FPGA require optimized HW-SW interfaces for data and control handling besides the generation of synthesizable HW project and compilable SW project descriptions. This work addresses these challenges of generation of HW drivers to enable the SW applications to efficiently utilize the available SoC resources.

1.3. Contributions

This work presents a novel approach for a SW-defined SoC design flow. The core ideas of the method are, (1) encode knowledge exchanged between the IP supplier and the SoC architect in machine-readable formats, (2) provide automation tools that can use this encoded knowledge to automate IP integration, (3) use model-based approaches and code generation to support any target HW prototyping platform. In order to be independent of any custom models, the method binds to industrial standards such as the IP-XACT IEEE 1685-2009 (IEEE 2009) and SW definition of the SoC functionality based on C/C++.

Fig. 1.3 provides an overview over the proposed approach. It starts at the side of the IP supplier, who generates a HW-accelerated software library (HASL). The HASL includes the hardware IP implementations, e.g., in Verilog or VHDL (.v/.vhdl), the interface descriptions using IP-XACT, and HW driver functions. As the key idea of this approach, it proposes to additionally encode IP-integration knowledge using a set of integration rules. The IP supplier encodes his strategy of efficient IP integration using these IP-integration rules. Using the declaration of SW functions for the application and the integration rules configurable HW drivers and simple schedulers are automatically generated. Only basic customization steps to configure the values of the control registers in the generated HW drivers are required from the IP supplier. The HW-SW interface between the HW IPs in the library and software functions is provided in the HASL. The

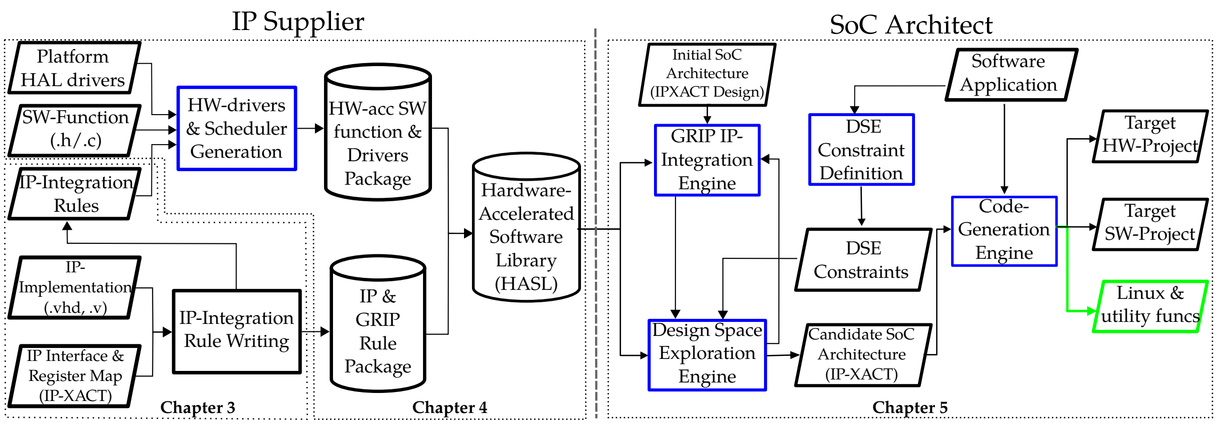

IP Supplier
SoC Architect
Platform HAL drivers
SW-Function (.h/.c)
IP-Integration Rules
IP-Implementation (.vhd, .v)
IP Interface & Register Map (IP-XACT)
HW-drivers & Scheduler Generation
HW-acc SW function & Drivers Package
IP-Integration Rule Writing
IP & GRIP Rule Package
Hardware-Accelerated Software Library (HASL)
Chapter 3
Chapter 4
Initial SoC Architecture (IPXACT Design)
GRIP IP-Integration Engine
Software Application
DSE Constraint Definition
DSE Constraints
Design Space Exploration Engine
Candidate SoC Architecture (IP-XACT)
Code-Generation Engine
Target HW-Project
Target SW-Project
Linux & utility funcs
Chapter 5

Figure 1.3.: The contributions of this work to bridge the knowledge gap between the IP suppliers and the SoC architects using the GRIP flow

result is a set of SW functions that have the capabilities to run hardware-accelerated using the available HW IPs.

The HASL forms the basis for bridging the gap between the IP supplier and the SoC architect. The SoC architect defines an application using the hardware-accelerated SW functions provided by the HASL. This SW definition for the SoC does not include any HW-related information. Based on this SW-defined SoC, an automatic IP integration is performed. In the proposed IP integration, first, an initial SoC architecture and the rules are translated from IP-XACT descriptions to their graph representations. Then, using the formal principles of graph grammars, also known as graph rewriting (Ehrig et al. 1981),(Geiss et al. 2006), new HW IPs can be integrated into the SoC. The SoC modeling in graph-space works transparently with IP-XACT, i.e. the IP-XACT SoC description can be read-in or generated from the graph domain at any stage of the design flow. All SoC modifications are transparent and can be backtracked or undone. After each architectural change, the generated IP-XACT SoCs are verified for design correctness. The consistency of bus protocols and signals is maintained during the IP integration. An iterative application of the rules spans a design space exploration (DSE) tree based on the IPs in the HASL. The proposed design flow enables a high-degree of automation for the SW-defined SoC design. From here, we will call this proposed flow the GRaph-grammar IP integration (GRIP) flow. In this work, an electronic design automation (EDA) tool is implemented to support the GRIP flow, called the GRIP tool.

The GRIP tool is implemented model-based such that code generation techniques can be used to generate SoC implementation files for any hardware platform. The generated files include SoC top-level descriptions for the HW project, the HW drivers as well as a task scheduler for the SW project. The SW definition in C/C++ does not need to be adapted, but the underlying HW driver functions are configured based on the SoC architecture. The GRIP tool supports at the moment the generation of HW and SW projects for the Xilinx toolchain. With these project files, bare-metal and Linux-based SW applications with HW acceleration can be compiled for the Xilinx Zynq FPGA (ZedBoard 2012).

The main contributions of this work are:

1. The formalization of IP-integration knowledge in a set of IP-integration rules. These rules describe the step-by-step IP-integration knowledge of an IP supplier in a machine-readable form. They are described in the widely accepted IP-XACT standard. The rules can encode structural modifications desired during the IP-integration process. These rules are used in the GRIP flow to automate the IP-integration process.

2. The automation of preparing the hardware-accelerated SW library. The HASL bridges the knowledge gap between an SoC architect, a SW application developer and an IP supplier. The generation of HASL uses the available HW information of IPs and IP integration from IP packages to generate HW drivers and a simple scheduler, eliminating the need for an SoC architect to dig into the hardware details of an IP. The novelity of the approach is to use IP-XACT descriptions to generate hierarchical HW drivers in the HASL. The hierarchical implementation of the HW drivers helps to re-use the generated code, and hence eliminate the redundancy within the generated HW drivers code. The HW drivers are generated both to support a bare-metal application as well as a Linux OS based application.

3. The GRIP engine to automate the IP integration in an SoC using the IP-integration rules available via HASL. The tool uses model-based languages and graph grammar algorithms for implementation. At the first step, the IP-XACT description of the input SoC design and the integration rules are transformed to the graph domain. Graph transformations, corresponding to structural modifications in the SoC, are obtained by applying graph grammar rules. One novel contribution is to represent an IP-XACT SoC with a corresponding graph representation in its complete and contained form. This further helped to automate IP integration, HW drivers generation, and SoC DSE by utilizing graph grammar.

4. The interactive search using the design space exploration (DSE) tree spanned by iterative execution of rules, and thus an efficient design space exploration becomes possible in an automated way. The size of DSE tree grows exponentially to the number of available rules in the HASL. In order to restrict the exponential growth of the DSE tree, DSE constraints can be applied motivated by the data flow of the SW application. These constraints help prune non-optimal SoC candidates for the target software applications. In the interactive DSE, an SoC architect can also manually direct the step-by-step design exploration and back-trace undesired structural changes.

5. The code generation support for HW drivers and the SoC design files for the Xilinx Zynq FPGA SoC platform. In the code generation, the platform-independent IP-XACT SoC designs are transformed to the target platform-dependent descriptions. It is an intricate step at which the generated HW drivers in HASL are associated with the SW application using the configurations generated from the IP-XACT SoCs. The final outputs of the code generation engine are the SW project and HW project, which are respectively cross-compiled and synthesized for the target platform using the platform-specific toolchains.
6. The last contribution of this work is to use the GRIP tool with feedback of performance estimations of the generated SoCs. This is to dynamically guide the GRIP tool for SoC architecture exploration. This is demonstrated for a problem of performance monitoring of an SoC prototype on an FPGA. The insertion of hardware monitors into an SoC for performance monitoring constrained to available FPGA hardware resources was formulated as a bin-packing problem. The GRIP tool generated the minimum number of SoC candidates with hardware monitors required for comprehensive SoC performance monitoring.

1.4. Publications on this Thesis

The key contributions of this work have been published in the following three conference publications and one journal paper,

1. Jassi, M., Mueller-Gritschneder, D. & Schlichtmann, U. (2015): GRIP: Grammar-Based IP Integration and Packaging for Acceleration-Rich SoC Designs, Proceedings of Design Automation Conference (DAC 2015) S. 383–397
2. Jassi, M., Bordes, B., Mueller-Gritschneder, D. & Schlichtmann, U. (2015): Automation of FPGA Performance Monitoring and Debugging Using IP-XACT and Graph-Grammars, 2015 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), S. 1–4
3. Jassi, M., Sharif, U., Mueller-Gritschneder, D. & Schlichtmann, U. (2016): Hardware-Accelerated Software Libraries Drivers Generation for IP-Centric SoC Designs, 2016 International Great Lakes Symposium on VLSI (GLSVLSI), S. 287–292.
4. Jassi, Munish, Hu, Yong, Mueller-Gritschneder, Daniel & Schlichtmann, Ulf (2018): GRIP - Graph-Grammar-Based IP-Integration - An EDA Tool for Software-Defined SoCs, ACM Trans. Des. Autom. Electron. Syst. 23(3): 40:1–40:26.

The first publication (Jassi, Mueller-Gritschneder & Schlichtmann 2015) focused on the SoC HW design automation, which presented the ideas of IP packaging, automated IP integration and SoC design space exploration. These are the contributions 1, 3, and 4 of the previous section. In the second publication (Jassi, Bordes, Mueller-Gritschneder

& Schlichtmann 2015), the SoC HW design automation flow was utilized to solve the problem of FPGA-based SoC performance monitoring. This work was awarded as *the second best EDA tools paper* of the SMACD'15 conference. This is the contribution 6 in the previous section. The third publication (Jassi et al. 2016) focused on the automation of HW drivers and HW-SW interface generation, which are the contributions 2 and 5 of the previous section.

This thesis combines these published works and presents a bigger picture as the GRIP flow for SW-defined SoC design. The thesis presents the proposed GRIP flow as a SW-defined SoC design methodology, and the associated EDA tools developed in this work. The fundamental ideas of this big picture are also presented in the fourth publication (Jassi et al. 2018). The GRIP flow is demonstrated on two computer vision (CV) applications case studies. These applications resemble the complexities of real-world use cases. In both applications, the target SoC captures video frames from a physical camera and perform CV tasks on the ARM Cortex-A9 CPU (ARM 2016). For the DSE, a few of the CV tasks are progressively transferred to dedicated HW accelerators. The results show a 4x to 150x processing time improvement with HW-accelerated image processing when compared to pure SW processing for different case scenarios with and without using an operating system.

1.5. Organization of the Thesis

The other chapters of this work are structured as follows. Chapter 2 discusses related work and background. Chapter 3 discusses the IP packaging for the GRIP flow. Chapter 4 discusses the generation of HW drivers and HW-SW interface for the HASL. Chapter 5 describes the automated IP integration and DSE targeted on the prepared HASL. Chapter 6 discusses the two CV case studies. Chapter 7 describes the FPGA-based SoC monitoring problem. Finally, Chapter 8 concludes.

2. Background

In this chapter, we will discuss previous contributions related to this work. We will also overview the mathematical principles of graph grammars and existing frameworks for model-based designs, which are utilized in this work.

2.1. State of the Art

The existing research works related to this thesis can be broadly segmented into the following categories: model-based design, the IP-XACT standard and IP reuse methodologies, HW-SW interface and HW drivers, SoC design space exploration, and FPGA bring-up and system performance estimations. Some of the key related contributions are presented below.

Model-Based Design: In order to support VLSI design with progressive refinements over various design abstractions, Gajski and Kuhn (Gajski & Kuhn 1983) proposed the 'Y' chart, with corresponding computer-aided design (CAD) tools to support the model refinements. This approach is generalized and further standardized by the Object Management Group (OMG) (Lecomte et al. 2011) for the Model Driven Architecture (MDA) approach (OMG - MDA 2003). The MDA approach describes using platform-independent models (PIM) in software development, and later refinements to platform-specific models (PSM). These models are meant to be defined by using OMG standards, like Unified Modeling Language (UML) (ISO 2004), MetaObject Facility (MOF) (OMG - MOF 1997), and Common Warehouse Metamodel (CWM) (OMG - CWM 2003). Meanwhile, UML originated as a formal general-purpose modeling language, and was accepted as an OMG and ISO standard. These model-based approaches were initially utilized for software development, and were lated evolved for HW designs. Over time, many UML profiles have been developed for HW designs, some of the popular ones include,

- SPT (OMG - SPT 2005) - Schedulability Performance Time,
- MARTE (MARTE 2007) - Modeling and Analysis of Real-Time Embedded systems,
- UML4SOC (OMG - UML4SOC 2005) UML for SoC,
- DIPLODOCUS (Apvrille et al. 2006) profile for fast SoC simulation and verification,
- Tampere University of Technology (TUT) profile (Kangas 2006) describes SW applications, HW platforms, and mapping of them.

A lot of tool chains have been developed to support various UML profiles. Papyrus (Papyrus 2016) provides a development platform for the MARTE profile. ACCORD/UML (Gérard et al. 2002) is developed to support real-time embedded systems using UML. In (Ecker et al. 2008), the author used a UML profile for HW-SW interface code generation. A work from D. Aulagnier et al. (Aulagnier et al. 2009) targets to develop SoC using MARTE.

All these UML profiles above are restrained by limitations of UML to describe HW-SW systems and Systems engineering semantics. Later, UML was extended with two additional diagrams to form the SysML profile (OMG - SysML 2008) (requirements diagram, parametric diagram), as a UML profile for the Systems engineering. However, there are still key problems associated with using UML for HW designs (Ecker et al. 2009),

1. It is very tedious and lacks expressiveness to include the IP interface details in a UML profile, like register memory map, bus and signal interfaces. This makes the UML profiles impractical to use for IP exchange.
2. The UML profiles that describe HW designs as the PIM lack enough details to be refined to HW models for the implementation on a target HW platforms.

The model-based approaches are seen as the most advanced methods for IP integration. An overview of such works is presented in (Vincentelli et al. 2009). Two tools, Ptolemy (Brooks 2005) and Metropolis (Balarin 2001) are described, which focus on model-based and platform-based design. Yet, these approaches still require the SoC architect to understand the inherent trade-off and requirements of third-party IP components. Thus, these methods are very valuable for system design and early prediction but do not solve the discussed challenges of IP reuse.

IP-XACT and IP reuse: IP-XACT originated to describe IP interfaces in the user-friendly XML format to promote IP reuse. IP-XACT is a widely accepted IEEE 1685-2009 (IEEE 2009) standard for electronic design descriptions. It is established by the Accellera Initiative (Accelera 2009). The initial industrial use cases of IP-XACT are presented by (Kruijtzer et al. 2008), which illustrates using IP-XACT for integration of IP cores and design verification. A few works have also proposed vendor extensions to IP-XACT standard to include timing information (Khan et al. 2008), software related features (Kamppi et al. 2013), and reconfigurable computing (Nane et al. 2011). The work from (Ochoa-Ruiz et al. 2011) used IP-XACT for dynamic partial reconfiguration systems. That work also presented using IP-XACT component interfaces meta-data information for verifying the interfaces. The work from (Liu et al. 2012) comes close to ours, it explored describing IP components using SystemC and later generating high-level synthesis on these components for complete implementation. This is a top-down flow that cannot make use of third-party IP components. There have been few works focusing on the UML profile to IP-XACT transformations, and vice-versa. The work presented by C. Andre et.al. (André et al. 2008) used the MARTE UML profile to model IP-XACT descriptions.

IP-XACT is an accepted SoC modeling standard, and is continuously being revised, IEEE-1685-2014 (IEEE 2014) (revision to IEEE-1685-2009 (IEEE 2009)). By using IP-XACT as interface to the GRIP tool, the tool benefits from the advantages of this standardization.

SoC Design Space Exploration: The earlier works on SoC optimization focused on heuristic-based methods. The works from (Li et al. 2005) and (Ascia et al. 2004) tackled this problem using genetic algorithms. The works from (Ferrandi et al. 2010), (Bhattacharya et al. 2010), and (Beltrame et al. 2010) respectively used ant-colony optimization, particle-swarm optimization, and Markov decision process for exploration. The work from (Lukasiewycz et al. 2009) used graph-based models for simultaneous optimization of the design, process mapping and communications in a system. Another work from (Givargis & Vahid 2002) developed the Platune tool that optimizes the SoC platform parameters.

The DIPLODOCUS profile is used by L. Apvrille et.al. (Apvrille et al. 2006) to perform system design space exploration. This profile benefits from the independent application and architecture modeling. On the UML application models, this work uses SystemC simulations and static analysis for fast estimations. It maps the application models on the architecture models, and later refinements to the final implementation.

The work presented in (Kamppi et al. 2013) is also related to the presented work in this thesis. It proposes extensions to IP-XACT to include SW changes for any HW change. Another related work from (Herrera et al. 2012) proposed IP-XACT extensions to include additional information on SW mapping. The IP-XACT models are extracted from MARTE and executable performance models are generated targeted to SCoPE (Botella et al. 2010) for performance estimations.

One common thing that is lacking in the existing methods is the encoding of design knowledge and incremental structural optimization. In industrial SoC designs, SoC architectures are incrementally optimized from the previous generation SoCs. Various SoC architecture alternatives are explored using SoC prototying. This requires encoding of design knowledge for repeatability and incremental optimization. Additionally, the earlier works do not allow to encode the knowledge of IP integration and the SoC design generation for targeted HW platforms. This work facilitates all these requirements for IP integration in the industrial SoC designs. The SoC optimization targets an IP library and provides required HW & SW code generation from IP-XACT for enabling SoC prototyping on FPGAs.

Software-defined SoCs: The recent Xilinx Software-defined SoC (SD-SoC) development platform (Xilinx 2015) provides a comprehensive platform for performing automated HW-SW partitioning of designs targeted for Xilinx-FPGAs. The limitation of this platform is that it uses high-level-synthesis to implement IP components with fixed register-map for setting communication flags. This limits its usage to integrate custom third-party IPs and allows only restricted SoC architectural changes. The generated drivers limit the HW usage to few fundamental operations and are non-extendible. Table 2.1 compares the key features of the SD-SoC platform, DIPLODOCUS UML profile based DSE and the GRIP tool.

Features	Xilinx SD-SoC	DIPLODOCUS UML DSE	GRIP Tool
SoC Modeling Support	IP-XACT	DIPLODOCUS UML	IP-XACT
Targeted platform-specific model	HDL (FPGA)	SystemC	HDL (FPGA)
HW-SW Co-design	YES	YES	YES
HW drivers generation	YES	NO	YES
SW-defined SoC design space exploration	YES	NO	YES
HW-platform independent IP integration	NO	NO	YES
Third-party IP-library (IP exchange)	NO	NO	YES
Customized IP-integration schemes	NO	NO	YES

Table 2.1.: Comparison of the key features of the Xilinx SD-SoC, DIPLODOCUS UML DSE and the GRIP tool

HW-SW co-design: Most of the earlier works on the HW-SW co-design used custom specification language to describe the HW-SW partitioning, and used code generation for synthesizing the hardware and software projects. In the work from (Chou et al. 1995) a HW-SW co-synthesis tool, Chinook, is developed that uses HW-SW design behavior specifications, and it generates hardware drivers and resource allocation using the specifications. In (King et al. 2012), the HW-SW interface is described by using Bluespec codesign language, which also describes the HW-SW partitioning. There are other works that dealt with the HW-SW co-design for unix-OS-based systems from (Ryzhyk et al. 2009) (Chen et al. 2014) (Wang et al. 2003) (Katayama et al. 2000) (King et al. 2015). In contrast, this thesis does not define any new specification language to describe HW-SW partitioning. This work differs from earlier works by proposing encoding IP-integration knowledge in IP-XACT rules, i.e. the knowledge of HW-SW mapping and the SoC design space is encoded within the available rules in the IP library. Further, this work proposes pruning of the design space using the DSE constraints, and the SW drivers generation.

FPGA bring-up and system performance estimations: There exists a lot of research work on system performance estimations for SoCs at various abstraction levels. For FPGA-based SoC prototyping, fast FPGA bring-up is very important. The paper (Gries 2003) reviews the major developments in design space exploration till early 2000s. It describes the challenges for design evaluation as well as the simulation and analytic models for system evaluation used during early design phases. The paper gives an essence of utilizing evaluation at multiple abstraction levels for doing extensive design space exploration (DSE) for hardware systems. (Densmore et al. 2006) proposes a method for improved estimations from the simulation engines by annotating the FPGA generated performance analysis. In that work new system architectures are generated by permuting directly the FPGA-vendor system specification files. In contrast, this work proposes a fast FPGA prototyping by automating the SoC modeling, and a vendor-independent IP-XACT description for SoCs that can later be targeted on a specific FPGA.

Work from (Guo et al. 2008) looked into FPGA-based performance analysis for crypto IPs when used in the scope of a complete system. Additional Timer and Debug IPs were manually integrated into the system for estimating one set of performances and removed for the other set. For complex heterogeneous SoCs, requiring detailed analysis, a manual approach would become impractical. In Chapter 7, we will discuss about using bin-packing algorithms to perform design partitioning for SoC monitoring. (Coffman 2013) reviews bin packing algorithms. Recent work from (Lewis 2009) proposes the hill-climbing method for the minimum grouping problem. The GRIP work on FPGA performance monitoring, in general, does not focus on finding a better bin-packing algorithm, but to formulate the problem in-hand, finding an optimal solution and automating the FPGA bring-up process. In the GRIP approach, required analysis tasks can be formulated using provided interfacing methods and hence is extendible (limited by available FPGA resources).

Model-based languages: For the implementation of the GRIP tool, we have opted for using model-based languages and tools. The paper from (Paige et al. 2009) presents and compares the principles behind different approaches to implement domain-specific languages for model management and discusses the Epsilon model management framework. The GRIP tool is implemented using the Eclipse Modeling Framework (EMF) (Eclipse Modelling Framework (EMF) 2011), and the Epsilon (Epsilon 2006) family of languages and tools for model operations. This work uses the following modeling languages, Epsilon Object Language (EOL) (Epsilon - EOL 2006), Epsilon Validation Language (EVL) (Epsilon - EVL 2006), Object Constraint Language (OCL) (OMG - OCL 2006), and the Java FreeMarker template engine (Java - FreeMarker 2015).

2.2. Background

The foundation of this work lies in the principles of graph grammar and model-based tools, like OCL, EMF, OVL, & EVL. This section will discuss the theory of graph grammars and the model-based tools used in the GRIP tool.

2.2.1. Graph Grammars and Graph Rewriting

Graphs have been used for formulating structural information in computer science, mechanical & electronic engineering, and bioinformatics (Königseder & Shea 2014a) (Königseder & Shea 2014b) (Buchmann et al. 2012). Over time, multiple tools have

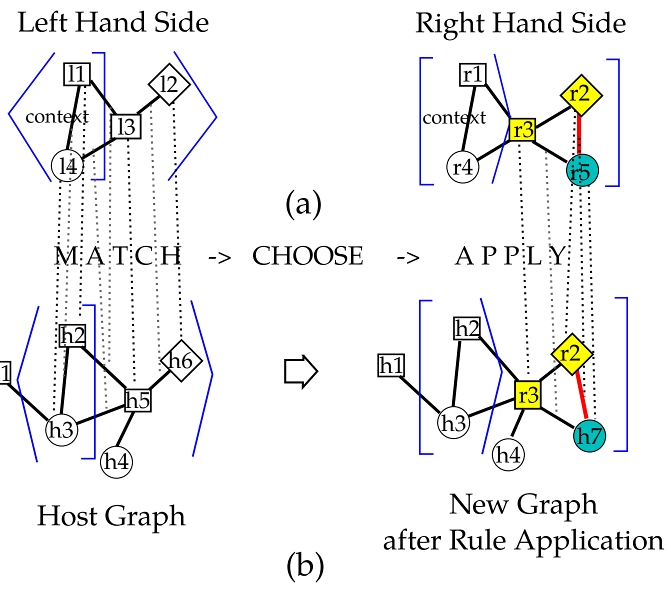


Figure 2.1.: (a) A sample graph-rewriting rule with the LHS and the RHS pattern; (b) Graph matching and application of the rule on the host graph

been developed to support graph-grammar-based modeling and structural transformations (GrGen tool v3.0 20011) (Booggie tool 2009) (eMoflon n.d.). Graph grammars are based on sound mathematics and offer a well-formulated way of describing evolving structural systems using graphs. The challenges of the evolving structures are: encoding the desired changes, localization of the changes, validation of the correctness of the final structures, and guiding & back-tracing the changes. Graph grammars can conveniently deal with these requirements. Yet, they require the target domain model to transform correctly and completely to the graph domain and vice-versa.

The structural changes in the graphs are achieved by using the principles of graph rewriting. Graph rewriting offers rule-based encoding of structural changes, and the application of rules on a host graph achieves those desired changes. A rule consists of a pair of graphs: a left-hand-side (LHS) pattern graph and a right-hand-side (RHS) rewriting graph. The graph rewriting is a three-step process, which is illustrated in Fig. 2.1 on a simple example. In the first step: match step, a match of the LHS is found in the initial host graph. A match is a sub-graph of the host graph, which has the same graph structure as the LHS, and whose node and edge properties match with the LHS. In Fig. 2.1 the nodes have properties indicated by their shape. In the choose step one of possible several matches is selected. Finally, in the apply step, the host graph is modified by replacing the LHS with the RHS to generate the new host graph. In the example, a new node *r5* is added and nodes *l2* and *l3* get modified to nodes *r2* and *r3* respectively. These modifications are applied on the node properties, in this case illustrated by the shape of the nodes. Thus, one needs also to know whether a certain node of the RHS reflects a new node or a modification of an existing node. It can also be seen that the nodes *h2* and *h3* are part of the matching process but remain unmodified. They form the *context* of the rule, which is kept unchanged during rule application as it is common to both sides. The context can be used to describe conditions under which the rule can be applied. We use a model-based graph rewriting approach described in Section 5.2. It allows to formally describe the matching process in the LHS as well as the required steps for node modification and creation in the RHS using model-based languages. Model-to-model transformations can be used to move across the domain meta-model, in our case among the IP-XACT domain and the graph domain.

2.2.2. IP-XACT Modeling of SoCs

A meta-model is a model of the model which represents the valid structure of the model. In this work, we have used the meta-model for SoC architectures described by the IP-XACT IEEE 1685-2009 standard. All the SoC architectures used in this work, whether those are the input SoCs or the generated output SoCs, are confined to IP-XACT. IP-XACT describes a system using XML files. An IP-XACT design object can be described using the available meta-models for component instances, bus and signal interconnections, library references, and vendor-specific customization. An IP-XACT design object can be seen as an SoC netlist. The IP-XACT component objects are described for their input and output bus ports and signals, parameters of configurable components, clock and interrupt signals, and memory-mapped registers and bit fields. An IP-XACT component object defines the interfaces of an IP component. The standardization of IP-XACT enables the portability of IP components across multiple vendors to support IP reuse.

Fig. 2.2 describes a simplified IP-XACT design and component modeling hierarchy as a UML diagram. An IP-XACT XML file describes either a design or a component under the *DocumentRoot* element. An IP-XACT design contains component instances, bus and signal interconnections, memory-map, top-level design IO ports, and design parameters. Each instantiated component in the IP-XACT design is described in its respective IP-XACT component instance file, and has a reference to a library component uniquely referred by the VLNV (vendor, library, name, version). The IP-XACT component instance describes the buses and signal pins I/O interfaces, registers and bit-field memory maps, and configuration parameters. The bus interfaces of an IP-XACT component refer to *BusAbstraction* abstractions via VLNV. The BusInterface describes a bundle of signal ports for a bus protocol. Further, the signal ports are described with their respective signal types, directions, and vector widths for their usage as Master, Slave, Mirrored-Master, or Mirrored-Slave. This hierarchical system description makes it suitable to accumulate design knowledge at multiple usage levels and realize reuse at different design abstractions. In this work, we have utilized this hierarchy of IP-XACT while implementing the GRIP engines for IP-XACT design validation and IP integration. It helped the implemented algorithms to abstract the design details at the signals and registers level to the buses and interfaces level, and hence speed-up the processes.

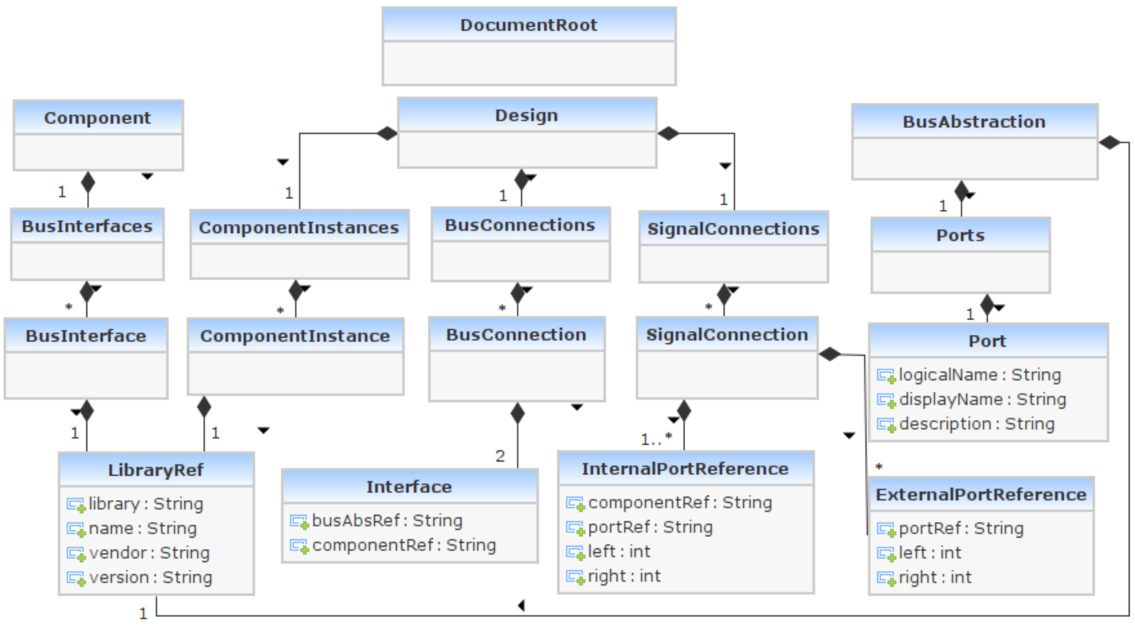


Figure 2.2.: Simplified UML model diagram for IP-XACT showing design and components hierarchies

2.2.3. Model-Based Design (MBD)

Model-based design (MBD) addresses the problems of designing complex systems by utilizing models that characterize continuous-time or discrete-time functional behaviors of complex blocks used in the systems. It has been instrumental in improving the productivity of designing complex engineering systems in previous years, especially in automotive, industrial and software engineering (Denney & Trac 2008). In MBD, the low-level implementations of system modules are abstracted by their high-level behavioral models. The models are complete to interpret the desired characteristics and leave away unnecessary details. By using MBD approaches, a designer can focus on integration of models to design a complete system, analyze and verify high-level system functionalities, rather on the implementation details. It helps finding system errors in the early design phases, rectifying and optimizing the system, and save time and effort spent on developing extensive software code and detailed system implementations. Later, after designing the complete system, the same models can be used with code generators to implement a system.

2.2.4. Eclipse Modeling Framework (EMF)

For the implementation of the GRIP tool, we have used the eclipse modeling framework (EMF). EMF is a powerful framework for model-based design, which provides a framework for interpreting, modeling, code generation and run-time facilities for developing new software tools based on structured data models. It is a Java-based platform and provides powerful utilities to handle models and XML files based on meta-models defined in the Ecore format. Ecore is the meta-model used by EMF to describe other models. Ecore is similar to an UML class diagram. EMF automatically generates the required Java classes and API to work with models defined by the meta-model. There are three types of code generation supported by EMF,

a. Model classes generation, it contains all the interface and implementation classes for the data model described by the meta-model.

b. Adapter classes generation, it contains the Java classes to adapt the model classes.

c. Editor generation, it provides the classes for Eclipse model editor.

EMF allows also to generate Ecore meta-models from XML Schema. An XML Schema describes the structure of an XML file. It defines constraints on the tags, elements and the content of the desired XML descriptions. In this work, we have used the XML Schema for IP-XACT published by (Accelera 2009) to generate the Ecore IP-XACT meta-model and API. Using these EMF generated IP-XACT meta-model and API, custom model-to-model transformations are implemented to the graph domain. In the graph domain, Ecore meta-models can be defined for describing nodes and edges and their configuration properties.

2.2.5. Model-Based Languages

The model-based approach for the SoC domain and graph domain enables to use formal model-based languages and tools, including Object Constraint Language (OCL), Epsilon languages and Freemarker for the implementation of the GRIP tool.

Object Constraint Language (OCL)

The Object Management Group (OMG) published the Unified Modeling Language (UML) standard (UML ISO/IEC 19505-1 2011) for general-purpose objects modeling, especially for software systems. UML provides a graphical way to describe and analyse software designs. The visual UML modeling is accurate and complete, and leaves less space for ambiguities and misinterpretations. One key element in UML is the class diagram, which describes the class objects and relationship between them. However, UML is limited in usage to describe constraints on the class attributes and methods. OCL is a specification language which is very powerful to query UML models and can be used to write rules to specify model properties to some legal values or ranges. It is released by the OMG and published as ISO/IEC 19507 (OCL ISO/IEC 19507 2012), as an extension to UML. OCL is a declarative language, i.e. it only specifies what computation to be performed, but without how that computation must be performed. This makes it a side effects free programming language, which doesn't modify the states of calling functions. OCL has precise semantics and can be used to specify invariants of objects, which must remain true through the lifetime of an object. OCL can also formally define constraints on pre- and post conditions on model operations and methods.

In this work, OCL is used together with EMF to implement an IP-XACT verification tool. It reads component definitions from the IP-XACT formated library and translates design rules into OCL rules. These rules are then checked to verify whether components are integrated legally in an SoC design under consideration. In Fig. 2.2, we can observe the class relationship for signal connections. The class SignalConnection has an associational relationship with multiplicity of one-to-many with the classes InternalPortReference and ExternalPortreference. The association is of type "composition", i.e. the InternalPortReference and ExternalPortReference classes can not exist independently of the class SignalConnection. Each of these reference classes contains the attribute of type of the class Port. However, this UML class diagram does not specify what are the legal signal connections, in other words, what are the legal attributes for the InternalPortReference and ExternalPortReference classes for a legal signal connection.

These legality constraints are specified by the OCL constraints. In the following are a few examples of the OCL invariant constraints,

```
     self.externalPortReference -> size() = 0 and
                          self.internalPortReference -> size() = 1
```

Code 2.1: OCL constraint checks for floating signal connections

```
self.model.ports.port->select(p:PortType|p.wire.direction =
   ComponentPortDirectionType::out)->collect(p:PortType|p.name)->
   size() > 1
```

Code 2.2: OCL constraint checks for the nets with multiple masters

Code 2.1 checks for floating nets in an IP-XACT design by evaluating if a signal net has only one connected port. Code 2.2 checks for the nets with multiple masters. It evaluates if a net has more than one connected ports with 'out' direction.

Epsilon Languages and Tools

Previously, we have seen the domain-specific richness of IP-XACT and EMF Ecore meta-model to model an SoC. Besides, the MBD also requires the model-management languages for constraining, modifying, validating and transforming the models. Fig. 2.3 shows the four key components required for the model-based design methodologies. These model management components interact with the model-handling framework during model design. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) is a family of languages and tools for model operations, model-to-model transformations, constraint validation, and code-generation. The Epsilon family of languages consists of Epsilon Object Language (EOL), Epsilon Transformation Language (ETL), Epsilon Validation Language (EVL), Epsilon Generation Language (EGL), Epsilon Wizard Language (EWL), Epsilon Comparison Language (ECL), and Epsilon Merging Language (EML). In our work we have extensively used EOL and EVL for the implementation of the graph re-writing engine.

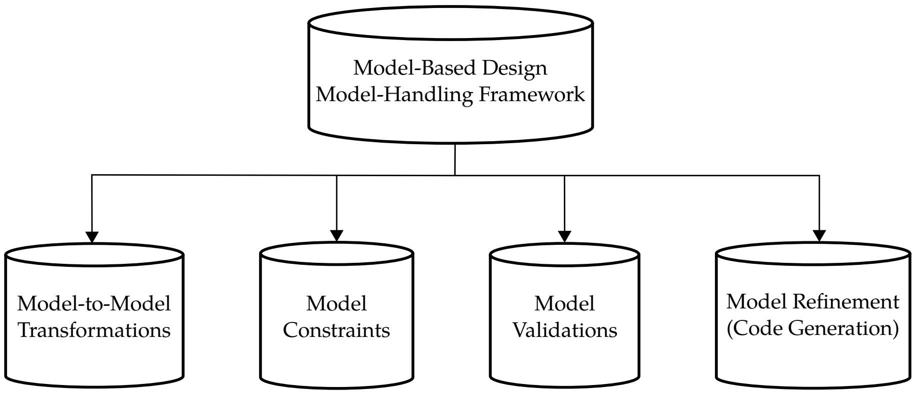


Figure 2.3.: Key components required for model-based designs

EOL (S. Kolovos et al. 2006) is a model management language which is used for modifying, creating and querying models. It is a meta-model-independent language extended from OCL. It overcomes the limitations of OCL, e.g. OCL can't be used for modifying, and creating new elements which is essential for model modification. Unlike OCL, EOL allows sequential statements for queries on models, besides it also supports debugging and error queries, model modification, and access to multiple models. In this work, EOL is used to describe the creation and modification of nodes during the apply step of graph rewriting. Code 2.3 shows an example OCL script to create a bus interconnection between a CPU component (*CPU_1*) and a bus (*bus_0*).

```
var interconnect = new InterconnectionType;
var interface1 = new Interface;
interface1.componentRef = "bus_0";
var interface2 = new Interface;
interface2.componentRef = "CPU_1";
interconnect.activeInterface.add(interface1);
interconnect.activeInterface.add(interface2);
InterconnectionType.all.add(interconnect);
```

Code 2.3: An example EOL script for adding a new bus interconnection.

EVL is a model validation language which uses constraints on a model. EVL (Epsilon - EVL 2006) inherits most of the OCL features, additionally it supports dependencies among the constraints. In this work, EVL is used to implement the graph matching step. Code 2.4 shows an EVL example script for the component instance matching, it specifies constraints to match the desired IP instance (component) name and library name. In the code, it can be seen that the `constraint: type` has a `guard` before `check`. This ensures that the `constraint: size` must be satisfied before checking the `constraint: type`.

An engine is provided by Epsilon to execute those languages in the EMF platform. It can read model objects from Java and modify them according to EOL scripts or return a boolean value showing whether they satisfy rules defined in EVL scripts.

FreeMarker Template Language

FreeMarker is a code generation engine to generate text outputs from templates and a Java object. Templates are composed of a mixture of sections, including the text, interpolation and FreeMarker Template Language (FTL) tag. The text section is printed to the output as it as. The interpolation section can print a calculated value, which can use both standard math and string operators provided by FTL and member methods of the input Java object. The FTL tag sections are used to provide programming language features, such as conditional statements, loops and containers. This template code generation engine is used to output the platform-specific files. Code 2.5 is an example FreeMarker template for generating EVL script as in the code 2.4. The template provide placeholders for variables identified by `'$'`, which get the values from Java function calls.

```
context EObject{
  constraint size {
    check : self->size() = 1
  }
  constraint type {
    guard: self.satisfies("size")
    check : ComponentInstanceType.all.size() = 1
  }
}

context ComponentInstanceType{
  constraint ref {
    check: self.componentRef.name = "bus_0"
  }
  constraint lib {
    check: self.componentRef.library = "soc"
  }
}
```

Code 2.4: An example EVL script for checking valid identifiers of a component.

```
<#-- the template input is an "ComponentInstanceType"-->
<#-- the template output is an EVL for "ComponentInstanceType" -->
context ComponentInstanceType{
  constraint ref {
    check: self.componentRef.name = "${getComponentRef().getName()}"
  }
  constraint lib {
    check: self.componentRef.library = "${getComponentRef().
   getLibrary()}"
  }
}
```

Code 2.5: An example FreeMarker template file for generating an EVL script file.

2.2.6. ZedBoard FPGA

In this work, FPGAs are chosen to be the target HW platform for SoC system prototyping. Specifically, the Digilent ZedBoard FPGA (ZedBoard 2012) is targeted by the GRIP tool engines. The GRIP code generation engine transforms the IP-XACT SoC descriptions to the HW specifications for the ZedBoard FPGA. The ZedBoard is an evaluation and development board based on the Xilinx Zynq-7000 All Programmable SoC. Besides, the board supports multiple IO interfaces: UART, USB, Ethernet, VGA, HDMI, PMOD - Peripheral Module interface, Push Buttons, and FMC - FPGA Mezzanine Card.

Xilinx Zynq Chipset

The Xilinx Zynq (Xilinx 2016) is among the first generation of FPGA chipsets that provide both a powerful hard-wired CPU and HW programmable fabric on the same chip. The architecture of the Zynq chip is split in two parts: Programmable System (PS), and Programmable Logic (PL). The PS part contains the ARM Cortex-A9 dual core processor, and few other hard-core IO interface IP components, including USB, UART, QSPI, GPIO, DMA, etc. The PL part is the FPGA part, which provides 106K FFs and 53K LUTs in the programmable fabric. The data transfers across the PS and PL parts can occur either through two General Purpose (GP) AXI-Lite interfaces, or four High Performance AXI4 interfaces. Additionally, there are other signal interfaces, including GPIOs, memory interfaces, interrupts, clocks and resets etc. Fig. 2.4 shows the block diagram for the Zynq chipset.

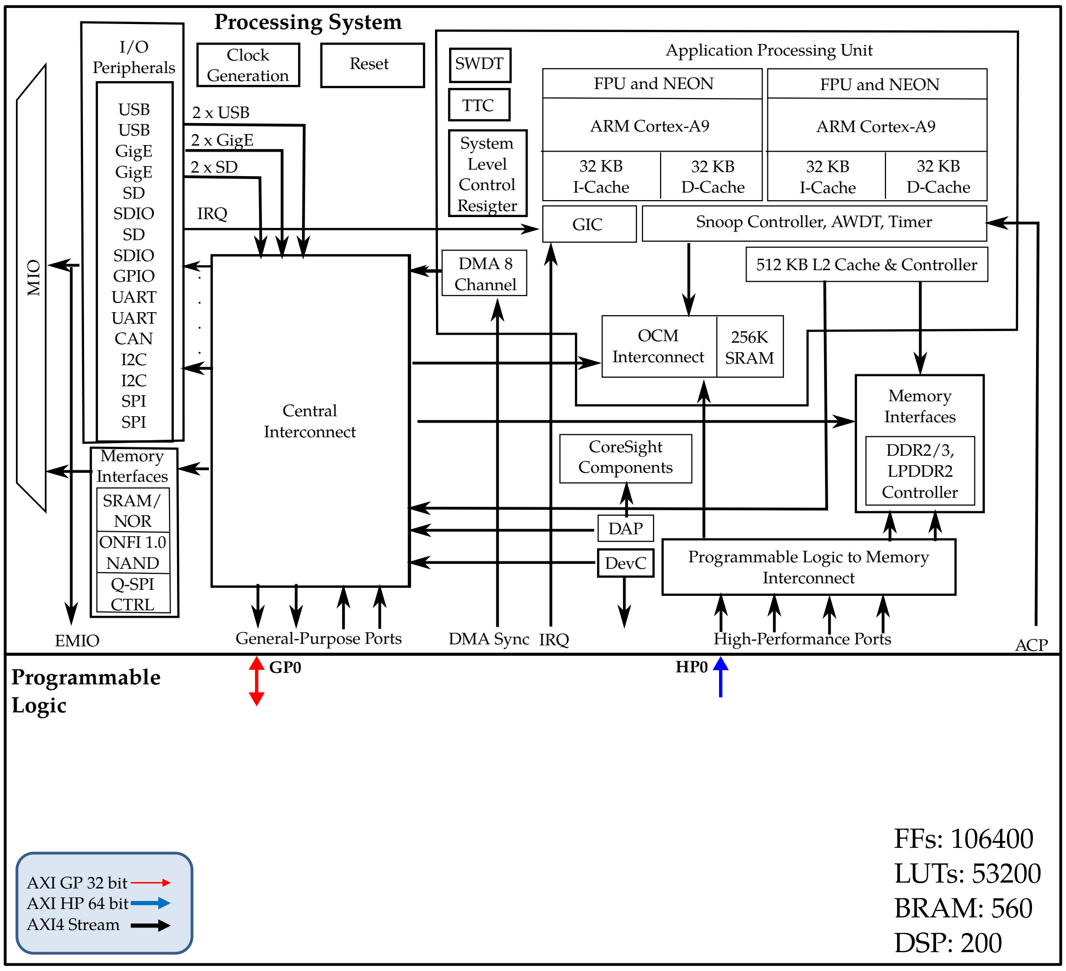


Figure 2.4.: Block diagram of the Zynq chipset with PS and PL parts

Xilinx ISE Toolchain for FPGA

Xilinx provides the software toolchain for the SoC modeling, synthesis, as well as software application development and compilation for the ZedBoard. The Xilinx Embedded Development Kit (EDK) is a combination of Xilinx Platform Studio (XPS) and the Software Development Kit (SDK). The XPS tool provides the development environment for designing the hardware project of the embedded system. It can be used to add desired IPs to an existing design and create interconnections between the IP components. The configurations of processor and other HW IPs, interconnection among these IP components and their address maps can be specified in XPS.

The SDK is an integrated development environment, complementary to XPS, that is used for C/C++ embedded software application development and SW binaries generation. SDK may also be used to debug software applications. SDK is based on the Eclipse open-source framework.

2.3. Conclusions

In this chapter, we discussed about the previous key research works related to the model-based designs, HW-SW co-design, IP reuse, IP integration, and the IP-centeric SoC design space explorations. The chapter compared the key contributions of the GRIP tool to the existing solutions. It advocated the need of more automation in the scope of SW-defined SoCs and SoC DSE to bridge the knowledge gap among the two actors in the SoC design: the IP supplier, and the SoC architect.

Lastly, we had an overview on the principles of graph grammars, and the existing tools and technologies for model-based design, which are utilized during the implementation of the GRIP tool.

3. IP Packaging for Automated IP Integration

"On the digital side, it means if traditionally we shipped a piece of RTL to the customer, you shipped documentation that goes along with it, the customer is going to figure out how to use this IP. So, they [customers] ask us, 'Hey, you ship us this IP, you ship us documentation, but we have no time to read this. Help us more.'" Johannes Stahl, Director System-Level, Synopsys, Inc.

3.1. Introduction and Problem

The increasing IP reuse has put a lot of burden on the efficient exchange of IP knowledge from the IP supplier to the SoC architect. For the SoC architect, an efficient IP exchange means to acquire sufficient knowledge to quickly integrate a new IP into an existing SoC in its most suitable configurations. At the same time, adapt the associated SW application to efficiently utilize the available HW resources. This problem becomes many-fold when multiple IPs are available in an IP library for an SoC DSE. A typical SoC can contain tens of IP components with thousands, even tens of thousands of registers and a number of SRAMs.

The IP supplier prepares an IP package in order to transfer an implemented IP and associated IP knowledge to the SoC architect. The IP package contains description of an IP in the hardware description language (HDL), hardware interface abstraction models (e.g. IP-XACT, SystemC), HW drivers, and documentations. A challenge for the IP supplier is to understand the system contexts in which the prepared IP will be used, and provide the required IP-integration information in the documentations. The SoC architect must extract the required information of valid bus protocols and interfaces for an IP, signal connections for interrupts, control and clock ports, configuration of control registers, data-communication handling and other relevant operational information. The standardization of IP interfaces using IP-XACT standard has facilitated to bridge the knowledge gap between the IP supplier and the SoC architect.

The main contribution of this chapter is the IP packaging to efficiently encode the IP-integration knowledge. The proposed packaging extends the current IP packaging with the additional information on the IP integration using the machine-readable integration rules, which are described in IP-XACT. The definition of rules uses the principles of the graph grammars. The chapter will discuss the rules definition using IP-XACT, completeness of the rules for different IP-integration scenarios, rules verification for their correctness and the rules preparation.

3.2. IP Packaging with IP-Integration Rule

In this chapter, we are proposing a solution to overcome the hassle of extracting the IP-integration information from an IP package. Fig. 3.1 shows the traditional IP packaging, which contains RTL descriptions of an IP, IP interface descriptions in IP-XACT, and IP documentation. In the proposed solution, the IP supplier can describe IP-integration knowledge using IP-integration rules. Each IP-integration rule describes one incremental change of the SoC architecture. The SoC architect can use these IP-integration rules to automate the IP-integration. The rules must be easy to develop, hence the rules must utilize the existing SoC modeling standards and available modeling tools.

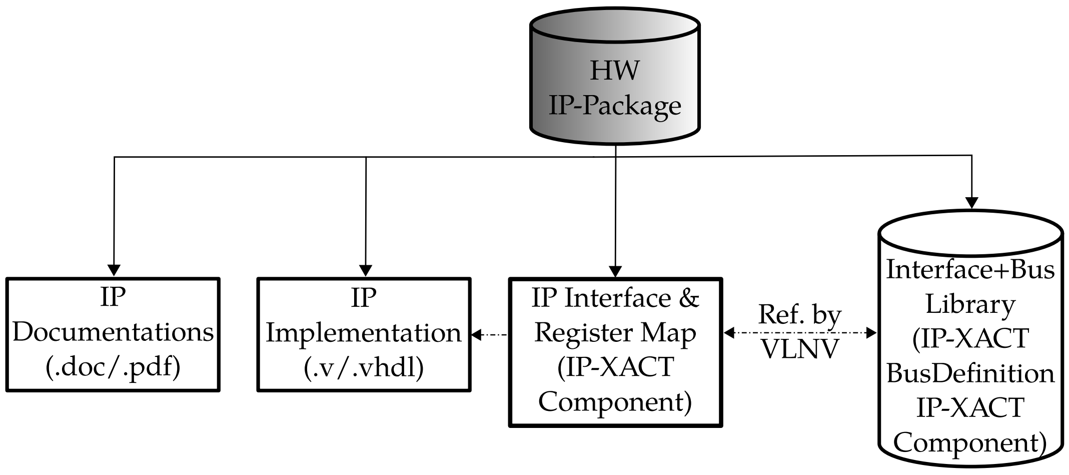


Figure 3.1.: Traditional IP packaging includes hardware descriptions, abstracted IP interfaces and documentation

3.2.1. What is an IP-integration rule?

The IP-integration rules are motivated by graph rewriting principles of the graph grammar theory (Sec. 2.2.1). Each IP-integration rule is composed of a left-hand-side (LHS) pattern design and a right-hand-side (RHS) pattern design. The LHS and RHS patterns are defined with IP-XACT design objects. The LHS pattern is matched to the input host SoC design. The match of LHS serves two purposes; first, it confirms the availability of required IP components in the host design, second, the found match localizes the position at which the change must be attained. The RHS pattern defines the desired changes, i.e. the IP-integration, on the host design. The IP integration is associated with changes in bus interfaces, signals and register configurations to other components on the host SoC, these changes are also specified by the RHS pattern. The application of a rule is a three step process, *I. Match, II. Choose, III. Apply.* In the match step, the LHS pattern is searched in the host SoC design. If the LHS pattern is found in the host design, and if multiple matches are found, as a next step, one of those matches is

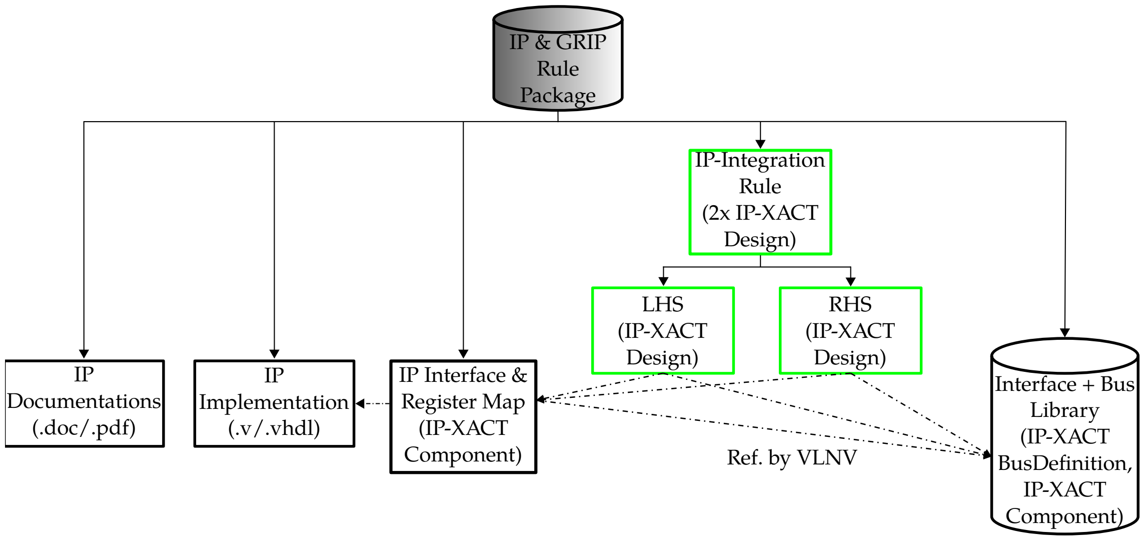

IP & GRIP Rule Package
IP-Integration Rule (2x IP-XACT Design)
LHS (IP-XACT Design)
RHS (IP-XACT Design)
IP Documentations (.doc/.pdf)
IP Implementation (.v/.vhdl)
IP Interface & Register Map (IP-XACT Component)
Ref. by VLNV
Interface + Bus Library (IP-XACT BusDefinition, IP-XACT Component)

Figure 3.2.: The proposed IP packaging includes the IP-integration knowledge using IP-integration rules on top of traditional IP package

chosen. In the final apply step, the chosen LHS match in the host design is replaced by the RHS pattern of the rule. The IP integration steps are explained in Chapter 5. Figure 3.2 shows the inclusion of IP-integration rules as an extension to the traditional IP packaging. The IP-integration rules must satisfy few necessary aspects of SoC design methodologies to make it suitable for SoCs. In Section 3.2.2, we will discuss the key characteristics of the IP-integration rules necessary for SoCs.

In IP-based SoC design, there are various structural changes of interest in an SoC. The IP-integration rules must assure the completeness of accommodating all possible SoC structural changes and scope of future extensions. Section 3.2.3 discusses the completeness of encoding structural changes using the IP-integration rules.

In the GRIP approach, all the SoC structural changes take place in the graph domain. One challenge here is to accurately model the SoC descriptions and IP-integration rules in the graph domain. The transformation functions for IP-integration rules must be a bijection from the IP-XACT domain to the graph domain. This bijection assures that the SoC design descriptions can be transparently transformed among the IP-XACT domain and the graph domain at any stage of the SoC structural transformations using the GRIP tool. The SoC description transformation from the IP-XACT domain to the graph domain is described in Section 3.2.4. These graph domain SoC descriptions are utilized later-on for verfication of IP-integration rules, as will be described in Section 3.3.

3.2.2. Characteristics of IP-Integration Rules

The IP-integration rules for SoCs must satisfy following characteristics to be suitable for automating IP-integration.

1. Standardization: The description of IP-integration rules must extend from the existing methodologies and standards for SoC implementation. In the proposed methodology, the IP-integration rules and the SoCs are described using the IP-XACT standard. It avoids the requirement to learn new modeling languages, also benefits from reuse of available IP-XACT SoC and IP interface descriptions from earlier projects. This is useful when modeling an SoC architecture one-time and targeting to multiple platforms during design refinement.

2. Completeness: The integration rules must cover all possible structural change scenarios in an SoC during IP-integration, and also be flexible for custom extensions. Since all the structural changes are performed in the graph domain, the completeness of structural changes require appropriate modeling of SoCs in the graph domain to abstract the heterogeneity of SoCs in the IP-XACT domain without losing any necessary information.

3. Localization: The desired structural changes during the IP integration are targeted at specific locations in an SoC w.r.t. existing IPs and bus-systems. The IP-integration rules must support to encode knowledge on localization in an SoC for attaining localized structural changes. In the proposed IP-integration rule definition, the left-hand-side (LHS) design pattern assures the localization of a desired change in an input host SoC.

4. Verification: One essential element while attaining a rule-based change in an input design is that the change shouldn't lead to an infeasible output design. The design-incorrectness in an integration rule will result in a structurally-incorrect final output design after application of the rule. Therefore, the integration rules must be verified for design correctness before applying them on SoCs.

5. Repeatability: The repeatability of a rule application is necessary to record the SoC structural changes, and iteratively apply those changes for progressive refinement of SoC architectures. Repeatability is also required to concatenate multiple rules for bigger architectural changes. Chapters 6 and 7 will describe a few case studies on SoC structural optimization using the IP-integration rules.

6. Handling: Writing, or editing an IP-integration rule should not be overhead for an IP supplier. Any new custom programming language to describe IP integration steps and HW-SW mapping would be an additional overhead for the IP supplier and SoC architect. In our work, all graph modeling and graph-grammar algorithms are kept hidden from the users, and all the user interfaces are via the IP-XACT standard.

All these characteristics are satisfied by an appropriate graph-domain modeling of SoCs, graph-rewriting principles to localize the desired change, an OCL-based SoC verification engine, choosing model-based approaches, and using IP-XACT for rules definition. The rest of this chapter and next chapters will describe these topics in detail. Section 3.2.3 describes completeness. The localization is achieved by applying graph grammars on graph abstractions of IP-XACT SoC descriptions, this is described in sec. 3.2.4. Section 3.3 describes verification of IP-XACT rules and SoC. Sec. 3.4 describes steps to write IP-integration rules for the GRIP tool. The repeatability and iteration of rules application is explained in Sec. 5.4.

3.2.3. Completeness of the Rules - Typical Cases for Rule Application

When considering the structural modifications in an SoC, there are global as well as local modifications. The global modifications include, addition, removal or modification of IP components, bus systems, and the interconnections. The local modifications are within the IP components, which include changes in IP parameters, IP-library component mapping, register memory map etc.. By the nature of these hardware modifications, most of them are independent of each-other and can be transformed into concatenation of simplified modifications. While, the interdependent hardware modifications where multiple IP components are required for the tasks processing and/or data communication, those must be described using a single IP-integration rule.

The specific IP-integration scenarios are formulated by the IP supplier, who has full knowledge about his IP components and their optimal integration. We have identified a range of typical SoC modifications listed below, which describe standard IP integration steps:

1. Adding a new bus system.
2. Adding an IP component or IP subsystem in parallel on an already existing bus.
3. Adding an IP component or IP subsystem in parallel on a new dedicated bus.
4. Adding a new IP component in pipeline.
5. Removing an existing IP component from an SoC.
6. Modifying bus or signal connections, without changing IP components.
7. Modifying IP component parameters, without any structural change.

The corresponding IP-integration rules for each class of changes are illustrated in Fig. 3.3. For each rules definition, a pair of IP-XACT design descriptions, one for the LHS and one for the RHS design pattern, must be prepared, e.g. using existing tools that provide a graphical interface such as (Kactus2 tool 2012). In the figure, the modifications in existing IPs from LHS to RHS are highlighted by yellow, while newly added IPs are highlighted by blue in the RHS pattern designs. The integration rules are integrated into the IP & IP-integration rule package (Fig. 3.2). The rules also allow to compose IP subsystems. An IP subsystem is an SoC part that consists of two or more IP components. As can be seen, most rules of Fig. 3.3 integrate IP subsystems composed of a hardware accelerator IP and VDMA (video direct memory access) IP.

The rule 1 in Fig. 3.3 encodes adding a new bus system, `axi3`, to an SoC which satisfies the corresponding LHS design pattern. During this modification, few signals of the CPU (`ps7_system`) are altered (CLKs, RST) to accommodate the new bus system, which is highlighted by the yellow color (for simplicity, signals are not shown in the figure). For the rule 2, integration of an IP subsystem consisting of a hardware accelerator (HA) IP connected to a video direct memory access (VDMA) IP is shown. The HA and VDMA IPs are connected via `axis` (AXI-streaming bus protocol) bus interface. The new subsystem is added to an existing bus-system in the LHS design (`axi3`). The rule 3 is an integration of a HA and VDMA IP subsystem on an additional dedicated bus system. This complex rule can be simplified as concatenation of rule 1 and rule 2. The complex rules can help to reduce the size of generated DSE tree. Rest of the rules are the manifestations of the descriptions above.

The IP instances in an IP-integration rule description (both LHS and RHS) are as described in the corresponding HASL. The IP instance mapping from the IP-integration rule to the HASL is defined during the IP-integration rule writing, and is essential for the rules verification.

3.2.4. Graph Transformation: IP-XACT to Architecture Graph (AG)

In the GRIP tool, the IP-XACT rules and the host design descriptions are first transformed to corresponding architectural graph representations (AGR). The architectural graph is an undirected graph. Each node and edge can store arbitrary parameters with different types, names and values. According to their data types, graph elements can be categorized into three types of nodes and two types of edges as follows.

1. IP component instance nodes: Each IP component instance is one node in the architectural graph (AG). Since in the IP-XACT standard, a bus-system is also instantiated as a IP component, each bus system is a node in the AG. These nodes contain common parameters that are the instance name and the component identifier, including the vendor name, the library name, the component name and the version (VLNV). Besides, each node also contains parameters that are specific to its instantiated IP component, such as component configuration, available bus interfaces and signal ports, memory spaces and so on. Code 3.1 shows a sample IP-XACT design file, in which IP components are instantiated as the `componentInstances` XML element. The IP component parameters are extracted from an IP-XACT component file, including VLNV and the IP component identifier under their respective XML elements (Code 3.2).

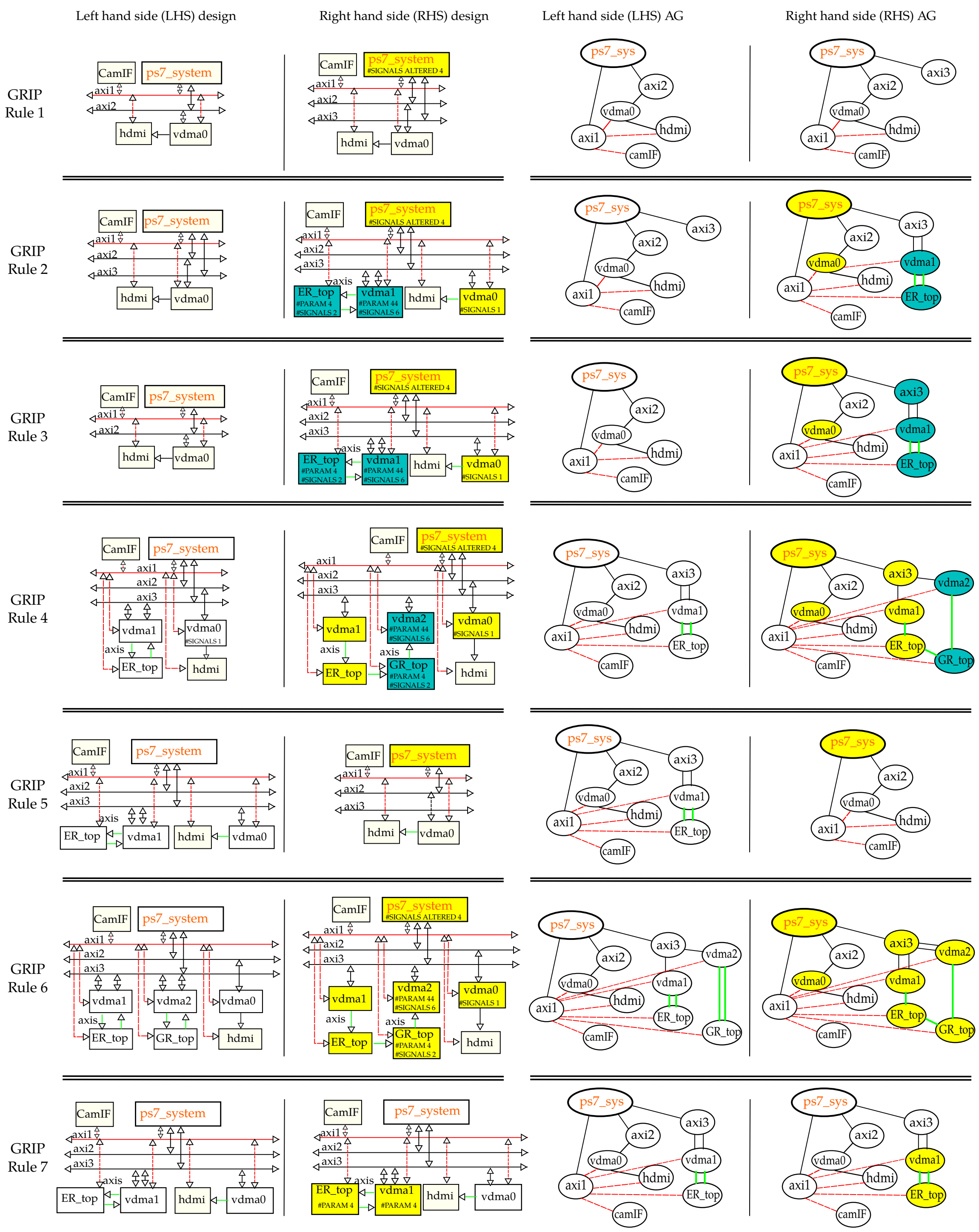


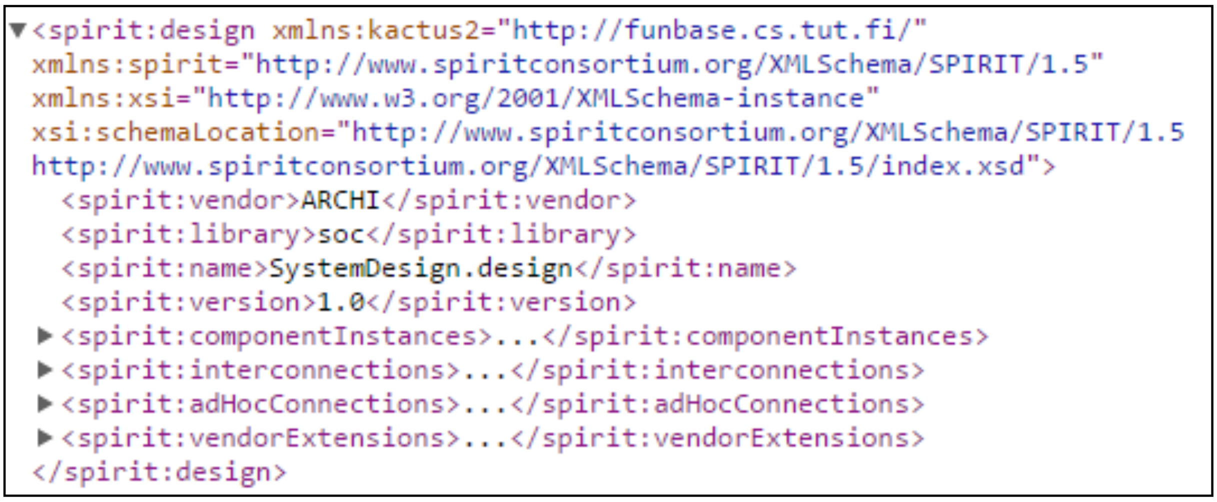
Figure 3.3.: Standard IP-integration rules for the possible structural modification scenarios.

2. Bus-interface nodes: Each bus interface associated to an IP component instance is one node. It is always connected with another bus-interface node and one instance node, which owns this bus interface. Each bus-interface node has only one parameter that is the name of the interface. This name should be contained in the available bus interfaces of the connected IP instance node. The IP instance node also stores properties of the bus interface, such as the interface protocol, data width, etc.. The bus interfaces of an IP component are available under `busInterfaces` XML element in an IP-XACT component file.

3. Signal port nodes: Both signal ports of each IP component and primary input-output external signal ports of an SoC are represented with nodes. These nodes have common parameters, including port name, MSB index and LSB index. The internal port node is connected to another port node and one instance node, while the external port node is only connected to another port node. The signal ports are available under `model` XML element in an IP-XACT component file.

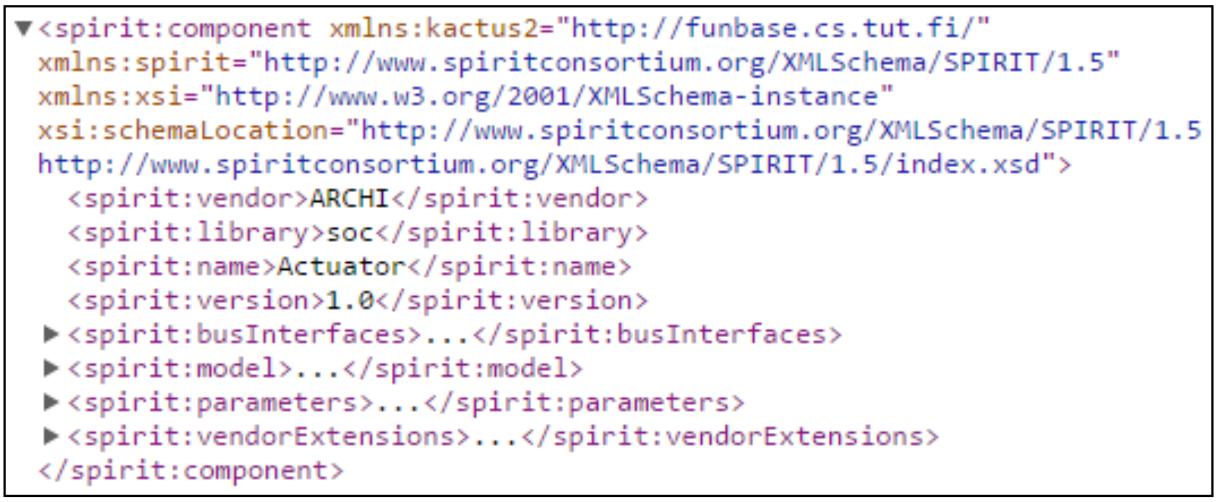
4. Logical connection edges: They are used to connect IP component instance nodes to bus-interface and signal port nodes. It is only a logical connection to store an IP component instance to which the ports and bus interfaces belong. These logical connections are available through IP-XACT component description files.

5. Physical connection edges: They are used to connect two bus-interface nodes or two signal port nodes. They represent physical nets in an SoC. Each of these edges also keeps a parameter for their respective net name. The physical connections are represented in IP-XACT design file under the XML elements, `interconnections` for bus interconnections and `adHocConnections` for signal connections.



```
▼<spirit:design xmlns:kactus2="http://funbase.cs.tut.fi/"
 xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
 http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd">
   <spirit:vendor>ARCHI</spirit:vendor>
   <spirit:library>soc</spirit:library>
   <spirit:name>SystemDesign.design</spirit:name>
   <spirit:version>1.0</spirit:version>
  ▶<spirit:componentInstances>...</spirit:componentInstances>
  ▶<spirit:interconnections>...</spirit:interconnections>
  ▶<spirit:adHocConnections>...</spirit:adHocConnections>
  ▶<spirit:vendorExtensions>...</spirit:vendorExtensions>
 </spirit:design>
```

Code 3.1: A sample IP-XACT design file showing the key XML elements



```
▼<spirit:component xmlns:kactus2="http://funbase.cs.tut.fi/"
 xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
 http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd">
   <spirit:vendor>ARCHI</spirit:vendor>
   <spirit:library>soc</spirit:library>
   <spirit:name>Actuator</spirit:name>
   <spirit:version>1.0</spirit:version>
 ▶<spirit:busInterfaces>...</spirit:busInterfaces>
 ▶<spirit:model>...</spirit:model>
 ▶<spirit:parameters>...</spirit:parameters>
 ▶<spirit:vendorExtensions>...</spirit:vendorExtensions>
 </spirit:component>
```

Code 3.2: A sample IP-XACT component file showing the key XML elements

Fig. 3.3 shows the graph representations of various IP-integration rules. Each of these node and edge classes are modeled with the EMF Ecore meta-model. This allows to use model-based languages for rewriting these nodes as will be shown in chapter 4. The architectural graph can be generated from the IP-XACT description using IP-XACT2AG transformation. As IP-XACT and AG nodes and edges are all modeled with the EMF Ecore, a model-to-model (M2M) transformation can be used. The architectural graph can be transformed back to an IP-XACT design without any loss using AG2IP-XACT M2M transformation, by using IP-XACT component descriptions from the associated IP library. Fig. 3.4 shows the M2M transformation of an example IP-XACT design to the corresponding AG. For simplification, the nodes and edges corresponding to signal connections are not shown in the AG. Further, a simplified AG is shown without the bus-port nodes. In the figure, the dashed-red, green and black edges respectively correspond to AXI-Lite, AXI-Stream and AXI4 bus protocols. All buses are transformed to EMF Ecore edge classes with their respective configurations.

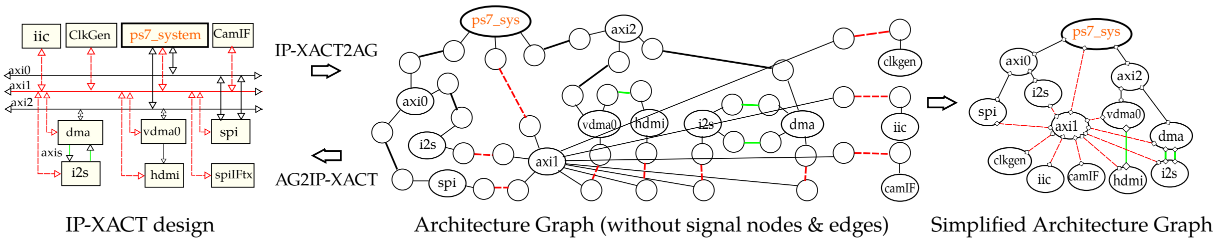


Figure 3.4.: Figure shows the model-to-model transformation of an IP-XACT design to the corresponding architecture graph

3.3. Verification of Integration Rules

Before the GRIP integration rules are applied to a host design, they must be checked for design correctness. The GRIP verification engine checks for the validity of LHS and RHS design patterns of each IP-integration rule. The valid LHS and RHS patterns assure that the structural changes by the application of the IP-integration rule will not result in an invalid output SoC design. A single incorrect rule when applied iteratively would result in consecutive generation of invalid designs. There are two types of verification under consideration, *static verification* and *dynamic verification*. The static verification of an IP-integration rule verifies the correctness of LHS and RHS IP-XACT designs before applying the rule. It is performed during the IP packaging and IP-library preparation stages in the IP-XACT design space. The dynamic verification is required during the process of rule application, which ensures the valid structural modifications are applied in the graph space.

3.3.1. Static Verification of IP-Integration Rules

The valid IP-XACT input SoCs and the IP-integration rules are necessary to ensure the correctness of SoC models in the graph space and further the correctness of encoding structural modifications. The static verification of the IP-integration rules includes the verification of both the LHS and RHS design patterns. The IP-XACT design verification tasks can be broadly segmented into two categories: IP component checks and interconnection checks.

IP component checks:

1. IP bus-signal ports verification
2. Floating IPs checks
3. IP design parameters verification

Interconnection checks:

1. Bus interconnection verification
2. Signal connection verification
3. Primary I/O signal connection verification
4. Special signals, clock, interrupt, power-ground checks

The IP component checks verify the correct instantiation of IP components in an IP-integration rule w.r.t. the corresponding component definitions in the IP library. The same checks are also applicable for correct instantiation of IP components in the SoC design. It verifies if all the bus and signal ports are correctly utilized in the design, and also check for floating IPs and the range of parameters values. The interconnection checks are for maintaining the bus and signal integrity of the design. It verifies the bus port connections (e.g. Master to Mirror Master), multiple drivers and floating nets, and handles the special signal nets.

In this work, the IP-XACT SoC design verification framework is developed using model-based languages, object constraint language (OCL). OCL is a specification language and can be used to write rules on legal values or ranges of XML model properties. This work, in my knowledge is the first effort to develop a design verification framework for IP-XACT using model-based formal methodologies.

```
private static void generateInterconnectionConstraints(<args>) {
    ...
    queries[i] = ocl.createQuery(helper.createQuery("self.
   busInterfaces.busInterface->select(i : BusInterfaceType | not i."
  +s +".oclIsUndefined())->collect(i : BusInterfaceType | i.name)->
   asSet()"));

  // generate constraints
  String[][] searchMap = {
    { "0", "0", "1", "1", "Master", "Slave" },
    { "1", "0", "0", "1", "Master", "Slave" },
    { "0", "0", "1", "2", "Master", "MirroredMaster" },
    { "1", "0", "0", "2", "Master", "MirroredMaster" },
    { "0", "1", "1", "3", "Slave", "MirroredSlave" },
    { "1", "1", "0", "3", "Slave", "MirroredSlave" } };
    ...

  cs  += "((self.activeInterface->at(1).componentRef='"
      + ci[Integer.parseInt(searchMap[k][0])]
      + "' and self.activeInterface->at(1).busRef='" + p1
      + "' and self.activeInterface->at(2).componentRef='"
      + ci[Integer.parseInt(searchMap[k][2])]
      + "' and self.activeInterface->at(2).busRef='" + p2
      + "') or (self.activeInterface->at(1).componentRef='"
      + ci[Integer.parseInt(searchMap[k][2])]
      + "' and self.activeInterface->at(1).busRef='" + p2
      + "' and self.activeInterface->at(2).componentRef='"
      + ci[Integer.parseInt(searchMap[k][0])]
      + "' and self.activeInterface->at(2).busRef='" + p1 + "'))" +
  " or ";
    ...
    }
```

Code 3.3: OCL constraints generation for correct bus interconnections in an SoC design

```
  private static void generateAdHoconnectionConstraints(<args>) {
      ...
switch(i){
case 0:{s = "out";break;}
case 1:{s = "inout";break;}
}
queries[i] = ocl.createQuery(helper.createQuery("if not self.model.
   ports.oclIsUndefined() then self.model.ports.port->select(p:
   PortType|p.wire.direction = ComponentPortDirectionType::"+s+")->
   collect(p:PortType|p.name)->asSet() else Set{'null'}->asSet()
   endif"));
              ...
String cs = "(self.name.substring(self.name.size()-2,self.name.size
   ()) = 'vdd') or "
    + "(self.name.substring(self.name.size()-2,self.name.size()) = '
   vss') or "
    + "(self.name.substring(self.name.size()-2,self.name.size()) = '
   vcc') or "
    + "(self.externalPortReference -> one(er:
   ExternalPortReferenceType|" + eos
    + ") and not self.internalPortReference -> exists(ir:
   InternalPortReferenceType|" + ios + ")) "
    + "or (not self.externalPortReference -> exists(er:
   ExternalPortReferenceType|" + eos
    + ") and self.internalPortReference -> one(ir:
   InternalPortReferenceType|" + ios + ")) "
    + "or( not self.externalPortReference -> exists(er:
   ExternalPortReferenceType|" + eos
    + ") and not self.internalPortReference -> exists(ir:
   InternalPortReferenceType|" + ios
    + ") and (self.externalPortReference -> exists(er:
   ExternalPortReferenceType|" + eios
    + ") or self.internalPortReference -> exists(ir:
   InternalPortReferenceType|" + iios  + ")))";
    ...
  }
```

Code 3.4: OCL constraints generation for correct signal interconnections in an SoC design

The required OCL constraints for static verification are dynamically generated according to the IP-XACT IP library, and design rule constraint templates. The generated OCL constraints are evaluated on the SoC design under verification. The code 3.3 shows an OCL constraint that checks for the correct bus ports usage as defined in the IP library. In this, the `queries` array is of size 4, and it collects all the Master, Mirror Master, Slave, and Mirror Slave bus ports of each IP from the IP library. The table `searchMap`

(2D array) defines the valid bus interconnections (eg. a Master to Mirror Master port). Then, according to this table, all the OCL constraints for valid bus interconnections are generated in the `cs` String. This check is iterative performed for all the SoC design bus ports. The codes 2.1 and 2.2 respectively are for floating nets and multi-master nets checks. As can be seen that verification checks are modular and independently target dedicated design checks. The checks are applied on SoC design descriptions in IP-XACT, and can be modified and extended as the IP-XACT standard evolves.

The code 3.4 generates OCL constraints to verify correct signal connections. The special nets, including power-ground nets are excluded from the signal interconnection checks. In the code, the `queries` accumulate all the `out` and `inout` ports of IP library components, and OCL constraints are generated for valid signal connections originating from these output ports. For all `externalPortReference` XML elements (IP-XACT SoC primary input-output ports), constraints are generated for valid `IN` direction to the `IN` direction of `internalPortReference` elements (signal pins on IP components).

Fig. 3.5 describes the OCL constraints generation and evaluation steps involved in the static verification engine. The inputs to the IP-XACT verification engine are the IP-XACT SoC description, IP library and design parameter constraints (optional). The engine generates all the required OCL constraints, and sequentially evaluates the generated constraints on the SoC design under test, as according to the earlier descriptions.

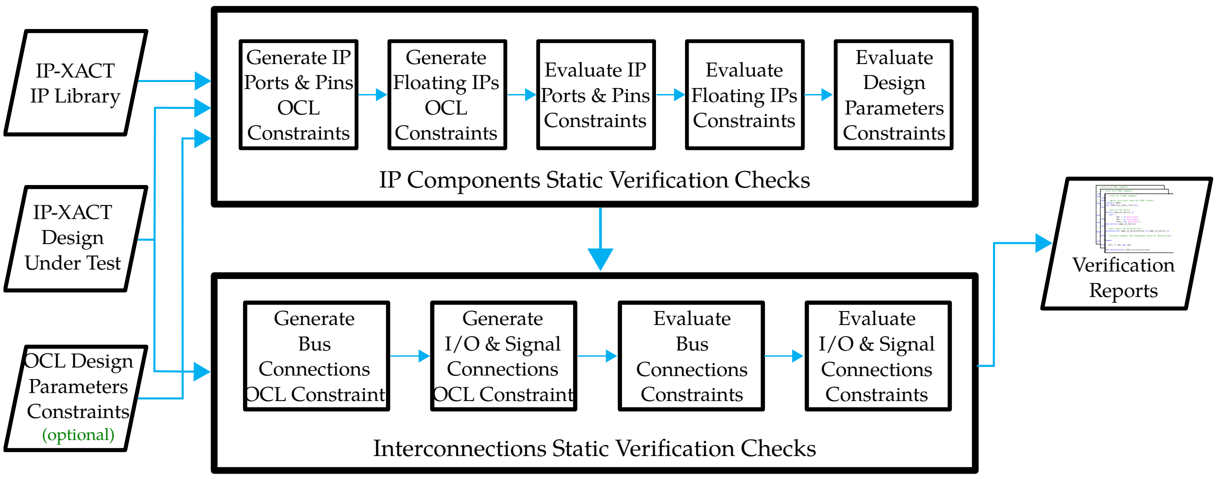

IP-XACT IP Library
IP-XACT Design Under Test
OCL Design Parameters Constraints (optional)
Generate IP Ports & Pins OCL Constraints
Generate Floating IPs OCL Constraints
Evaluate IP Ports & Pins Constraints
Evaluate Floating IPs Constraints
Evaluate Design Parameters Constraints
IP Components Static Verification Checks
Generate Bus Connections OCL Constraint
Generate I/O & Signal Connections OCL Constraint
Evaluate Bus Connections Constraints
Evaluate I/O & Signal Connections Constraints
Interconnections Static Verification Checks
Verification Reports

Figure 3.5.: Block diagram of IP-XACT static verification engine

3.3.2. Dynamic Verification of IP-Integration Rules

The dynamic verification confronts structural modifications during IP integration on a host SoC design. Though the modifications described by an IP-integration rule are independent of the host SoC design, when an IP-integration rule is applied to a host SoC design multiple design conflicts can arise because of newly added or modified IP components and interconnections. All these design conflicts must be probed and resolved for correct IP integration. The dynamic verification considers the IP-integration rule together with the host IP-XACT SoC design to which the rule is being applied. Some of the identified conflicts are listed below.

1. IP component or bus instance conflicts between the host SoC design and the RHS pattern of the rule.
2. Signal and bus net conflicts among the host SoC and the RHS pattern.
3. Ports or pins conflicts between existing bus or signal interconnections of the SoC and new interconnections in the RHS design.
4. Floating nets when disconnecting an interconnection from a pin or bus port.
5. Handling of special signals, interrupts, clock signals and power-ground nets.
6. Legalization of IP components and nets names for the final output IP-XACT SoC.

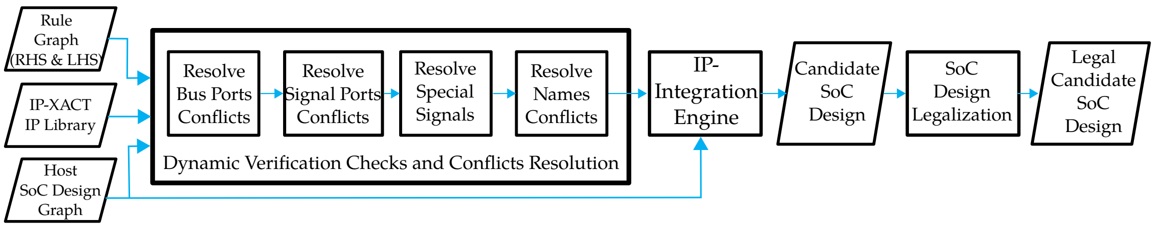


Figure 3.6.: IP-XACT dynamic verification and conflicts resolution during IP-integration

When using graph grammars for the IP integration, the structural changes as described in the RHS are performed on the host SoC, so only RHS pattern of the IP-integration rule is utilized during the dynamic verification. These mentioned conflicts are handled in the graph domain within the verification engine. The engine uses IP interface information available via the associated target IP library for various steps during verification. Figure 3.6 summarizes the dynamic verification and conflict resolution during the IP integration. At first, the RHS pattern of a IP-integration rule is compared to the host SoC design, which labels all the signals, nets and ports conflicts in the RHS pattern. Then, a copy of the RHS pattern is created and appropriately modified to resolve the labelled conflicts. The bus ports conflicts are resolved by iteratively reconnecting the conflicting connections to the available equivalent bus ports (reconnect if not used). If no equivalent bus port is available, the verification engine flags an error and the IP integration fails. For signal connections conflicts, the signal nets of RHS pattern and the SoC design are merged as long as each net has single driver pin (direction: `OUT`). If signal merging leads to multiple driver pins for a net, the engine flags an error and the IP integration fails. The *interrupt* signals are handled separately, by default, a new interrupt signal (in the RHS pattern) is considered as the least priority extension to existing interrupts in the host SoC. At the end of the generation of new candidate SoC, the final design is legalized for the bus, signal, IP instance logical name conflicts. The design legalization is performed hierarchically for faster runtime. First, all IP instances names are uniquified, all conflicts are resolved by adding unique post-fix integer. Second, the bus names are derived by concatenating, `<From instance name>_<From instance bus port name>_<To instance bus port name>_<To instance name>`. Next, the interconnection names are derived by concatenating, `<From instance name>_<From instance pin port name>`.

3.4. Writing an IP-Integration Rule

The IP-XACT standard provides the required XML elements to describe the IP interfaces and SoC designs, both for platform-independent as well as platform-specific descriptions. The platform specific details are used by the code generator engine to generate HW projects for the target HW platform. In this work, the GRIP tool is targeted to support the IP integration and SoC DSE on the Xilinx Zynq FPGA chipset. This section will describe the writing of IP-integration rules to target a Xilinx Zynq FPGA.

In the IP-integration rules, the required platform specific details are included in the `parameter` XML elements and the design descriptions. The Xilinx synthesis tool uses an SoC description in the *Microprocessor Hardware Specification (MHS)* format. Following are the key elements in an MHS description,

1. Top-level SoC I/O ports (direction, type, and size) and design parameters
2. IP component instances in an SoC with their parameters, clocks, memory map etc.
3. Bus-system instances with bus properties, clocks, bus ports etc.
4. Port-based bus and signal interconnections among instantiated IP components
5. Special signal nets, system clocks, interrupts, power-ground

The first step is to describe the bus protocols used in SoCs in IP-XACT. The target hardware platform (Zynq FPGA) uses ARM-based bus protocols, including Advanced Extensible Interface 4 (AXI4), AXI Lite, and AXI Stream (AMBA Specifications 2012). These bus protocols are described in their *busDefinition* IP-XACT files. The descriptions include the required logical signals, together with their respective properties (direction, size, type etc.) for all bus types (Master, Slave, Mirror Master, and Mirror Slave).

The second step is to describe the IP-XACT components required for the SoCs. The available bus definitions are used to describe the IP component bus interfaces, and are references by VLNV (vendor, library, name, version) XML elements. The IP component signal ports and parameters are described according to the Xilinx MHS specifications. Moreover, each IP component is also described for registers and bit-field memory map.

It should be noted that the Xilinx hardware synthesis (using an MHS file) doesn't require the register memory map. However, it will be required for generating hardware drivers and scheduler for software applications, as it will be discussed in chapter 4.

As a next step, these IP components are instantiated in IP-XACT design descriptions for LHS and RHS patterns. For the LHS design pattern, those IP components are instantiated that are prerequisites in a host SoC design for desired structural modifications. The RHS design pattern contains a set of IP components according to the desired structural modifications. In the IP-integration rules, the data communication as described by the bus interconnections in the LHS design are prerequisites for the rule application (edges in the graph space). However, signal connections of the LHS pattern are not set as the prerequisites, i.e. the signal connections of LHS pattern are not matched to signals of a host SoC during IP integration. The IP-integration still makes signal connection changes according to the differences between LHS and RHS patterns. The GRIP IP-integration engine makes modifications in a host SoC design according to the following guideline,

1. All changes in bus and signal interconnections from the LHS to RHS pattern, including addition, removal, or modification.
2. Addition or removal of IP components from the LHS to RHS pattern.
3. For the matching IP components between LHS and RHS patterns: all new IP component parameters in the RHS pattern are appends in a host SoC, and parameters with the identical *keys* are updated according to *values* from the RHS pattern.
4. The system I/O ports, and parameters are modified as from LHS to RHS pattern.
5. A new *interrupt* signal in the RHS pattern is appended as a least priority interrupt.
6. Register map for each IP component is kept unchanged.

3.5. Conclusions

This chapter presents one key idea of including the IP integration knowledge using IP integration rules in the IP packages. These IP-integration rules are based on the graph rewriting principles of graph grammar theory. An IP-integration rule has a left-hand-side (LHS) pattern design and a right-hand-side (RHS) pattern design, described in the IP-XACT standard. Each IP-integration rule describes one incremental change of the SoC architecture. By confining the IP data to the existing IP-XACT standard, it increases usability and portability.

The chapter discusses the completeness of SoC structural modifications using the IP-integration rules. It describes the associated GRIP engines to support rule-based IP-integration encoding, including OCL verification engine, IP-XACT to architecture graph representation (AGR) transformations.

4. Library Preparation - Hardware-Accelerated Software IP-Library

"Exacerbating the problem is the ever-present tension and lack of communication between chip architects and software developers, who generally have a limited understanding of what's going on inside the hardware. Yet these architects, developers, and hardware designers must work together as a team if the newest phase of the SoC revolution – platform-based design –is to become a reality for more than just a few large companies with vast resources.", Jack Shandle and Grant Martin, EEtimes.

4.1. Introduction and Problem

A few IP suppliers, like Xilinx OpenCV HW IP library (Stephen et al. 2015), supply multiple hardware IPs as a HW IP library for performing functional tasks of a specific application domain, e.g. computer vision domain. This domain-specific library also packages hardware drivers associated to the HW IPs. These HW drivers are utilized by the software developers to implement target software applications. While using these IP libraries, the software developers have to overcome a few challenges, as follows,

1. For utilizing available hardware resources, a SW developer must handle the initialization of the HW system, as well as setting the control register values of HW IPs, e.g. enabling or disabling processing, polling, data communication, synchronization etc.
2. While performing an SoC design space exploration (DSE) targeted to a domain-specific IP library, an SoC architecture goes through progressive structural changes. Adapting a target software application for each architectural change can be tedious.
3. If a desired HW-SW system utilizes an operating system (OS), another challenge is to reconfigure kernel-space drivers and build the OS kernel for each new SoC.
4. Another challenge is to take care of task scheduling on available HW resources, and appropriately handle the data processing control among the CPU and the hardware accelerator sub-systems.

To overcome all these challenges, the SW developer is required to understand hardware details of the target SoC system. It can be quite challenging for a software developer with insufficient HW knowledge to implement hardware interface and control functions.

In this chapter, we propose to resolve some of the mentioned software application development challenges by automatic generation of HW drivers by utilizing HW information available in the proposed IP packaging (in the previous chapter). The proposed stack of generated HW drivers of an IP library is divided into two levels: *hardware-access drivers* and *HW-SW interface*. The GRIP HW drivers generation engine uses IP-XACT design, component interfaces and register-map information to generate hierarchical HW drivers and a simple scheduler. In the generated HW-access drivers, at the lowest-level are hardware abstraction layer (HAL) drivers which are specific to a target hardware platform, then there are hardware IP drivers, and finally IP-subsystem drivers. The proposed HW-SW interface in this work associates the generated HW-access drivers to their corresponding functions in the software library. This HW-SW interface schedules an application task either as a SW task (available from SW library) on a CPU or on a dedicated HW accelerator. The accesses to multiple HW accelerating subsystems are handled by a generated Scheduler, which also forms a part of the generic HW-SW interface. These two layers of drivers - HW-access drivers and the generic HW-SW interface form the hardware-accelerated software library (HASL). The objective of this proposed HASL is to make hardware IP usage as simple as using a function call to the SW library.

The main contribution presented in this chapter is the hierarchical generation of HW drivers and the proposed generic HW-SW interface in compliance with the HW-access drivers from IP-XACT descriptions for both OS and non-OS environments. This enables SW applications to get automatically adapted to an underlying SoC architecture, especially during DSE.

4.2. Generic Hardware-Software Interface

In addition to the IPs and IP-integration rule packages, the HW-accelerated SW library also contains a package with the generic HW-SW interface and the HW-access drivers. These software codes need not to be written all manually but can be partly generated by the IP supplier based on the IP-XACT descriptions. This proposed automation of IP drivers code generation conceals HW details while providing access to HWs as simple function calls for the software applications. Separating the complete automation into the generic HW-SW interface and hardware-access drivers makes the solution modular and improves the scope of automation.

Fig. 4.1(a) shows that in the traditional software development a software needs to be adapted when an underlying SoC architecture changes. Using the proposed generic HW-SW interface, SoC architectural changes are accounted by automatically adapting the generic HW-SW interface and drivers (Fig. 4.1(b)). Figure 4.2 describes the flow diagram to access the HW resources from the SW application using the generated HW drivers. In this approach, a function call from the software application goes to the generic HW-SW

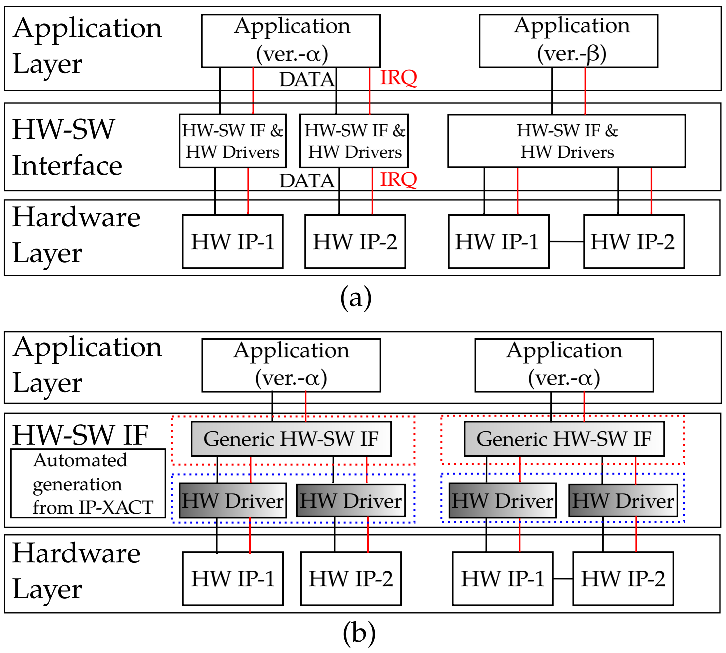


Figure 4.1.: a) SW adapting for the traditional HW-SW interface, b) proposed generic HW-SW interface, interface accommodates the HW changes,

interface, and the task scheduler performs the hardware checks to execute the SW tasks on a CPU or on a dedicated HW IP subsystem. It checks for the existence of targeted IPs or IP subsystem in an SoC, and if the IP subsystem is "not busy" for allocating an application task. If multiple IP subsystems exist in an SoC for an application task, the scheduler does round-robin-based task allocation. The generic interface provides an identical function call access to the SW as by the SW methods. For the target SoC, the HW-SW interface functions use generic configs (IPs base-address, interrupt IDs, register maps and drivers mapping for OS) obtained from the IP-XACT design descriptions.

In the Linux OS based systems, the virtual memory space is divided in two levels - kernel space and user space. The kernel space contains the HW drivers that are included and compiled together with the Linux kernel. The user space contains the SW applications and the user-space drivers, which are pre-compiled or compiled together with the SW application. The generic HW-SW interface forms a part of the user space.

Additionally, the generic HW-SW interface contains a few other functions for interrupt handling and initializing. These functions are used to associate the interrupts from HW subsystems to general interrupt controller (GIC) used in the SW application, in order to handle interrupt-based control synchronization (using interrupt service routines).

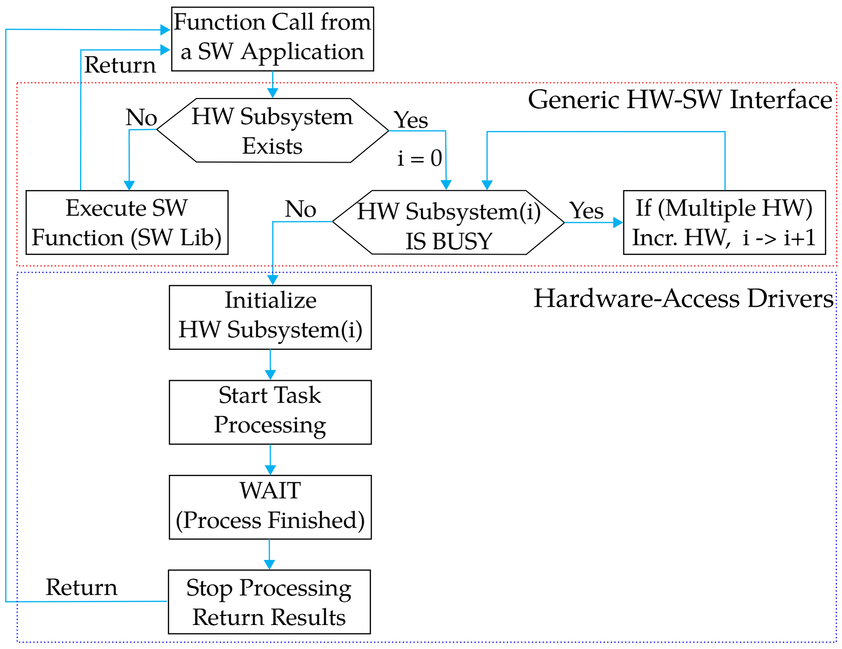


Figure 4.2.: Accessing either a hardware subsystem or software function from a software application through the generic HW-SW interface

4.3. Hardware-Access Drivers Generation from IP-XACT

The hierarchical system descriptions in IP-XACT makes it suitable to accumulate design knowledge at multiple usage levels and realize reuse at different design abstractions. The hierarchy of HW-access drivers functions generated by the GRIP tool is as described below.

- At the lowest layer are the hardware abstraction layer (HAL) and driver codes for accessing IPs and setting certain modes by writing to the register interface and setting specific bit fields. This layer contains the target platform-specific functions.
- The next layer contains IP drivers. This layer contains the functions to access the individual bit fields and registers of all the IPs contained in an IP library.
- A set of IPs that collectively processes an application task forms an IP subsystem. At the next layer are the drivers to access IP subsystems, which use the drivers of IPs in the subsystem.
- At the top layer are the generic HW-SW interface functions. It also has a simple scheduler that schedules an application task on an available IP subsystem, or on a CPU if IP subsystem is not available. These HW-SW interface functions are called by the application. Each function call corresponds to a task execution that can then be scheduled by the scheduling layer to an IP subsystem for HW execution or the CPU for SW execution.

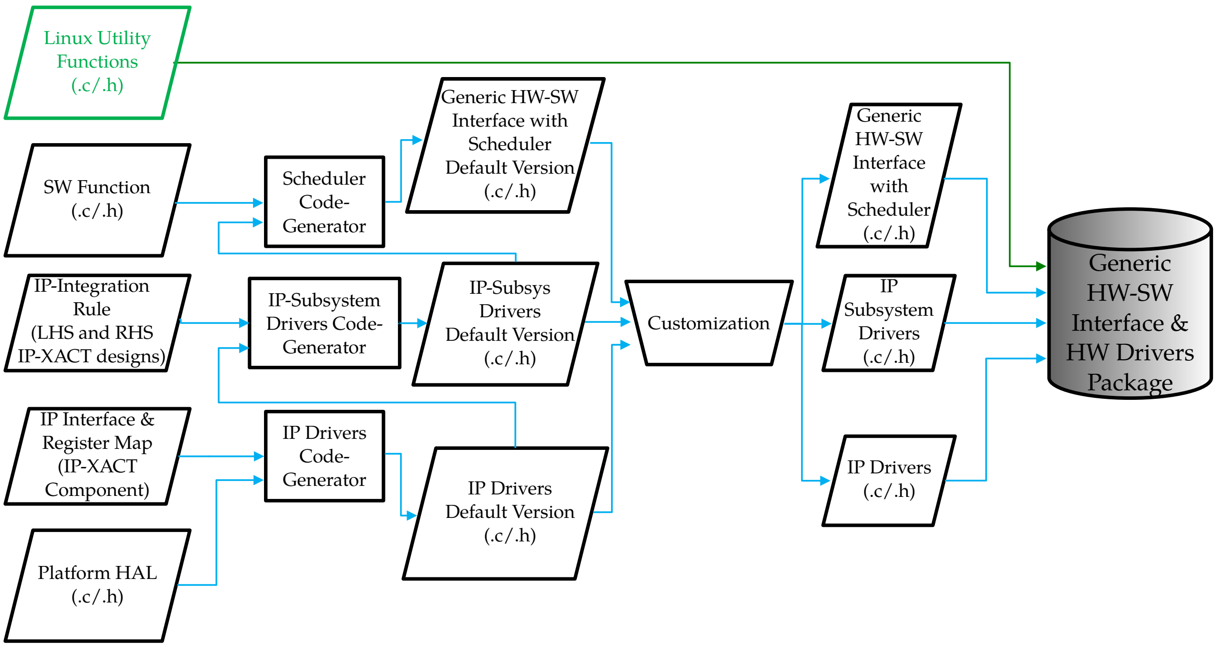

Linux Utility Functions (.c/.h)
SW Function (.c/.h)
IP-Integration Rule (LHS and RHS IP-XACT designs)
IP Interface & Register Map (IP-XACT Component)
Platform HAL (.c/.h)
Scheduler Code-Generator
IP-Subsystem Drivers Code-Generator
IP Drivers Code-Generator
Generic HW-SW Interface with Scheduler Default Version (.c/.h)
IP-Subsys Drivers Default Version (.c/.h)
IP Drivers Default Version (.c/.h)
Customization
Generic HW-SW Interface with Scheduler (.c/.h)
IP Subsystem Drivers (.c/.h)
IP Drivers (.c/.h)
Generic HW-SW Interface & HW Drivers Package

Figure 4.3.: Hardware drivers generation

Fig. 4.3 shows the SW code generation process. It can be seen that the inputs to the code generator engines are: IP-integration rules (LHS and RHS patterns), IP components available from IP packages, and SW functions for Linux utilities, SW task functions and target platform HAL. Based on the inputs, a default set of HW-access drivers is generated, which contains IP drivers, IP subsystem drivers, and a generic HW-SW interface with scheduler. Later, these drivers can be customized by the IP supplier to prepare the HW-access drivers package. These hierarchical drivers are explained in detail below.

The hardware abstraction layer (HAL) forms the lowest layer in the generated HW drivers, which varies for the bare metal and Linux OS based drivers. This layer contains read/write functions to the HW control registers and initialize the target HW platform. For the bare metal drivers, the HAL consists of HW drivers specific to the target HW platform, and are provided as the HW board support package (BSP) by the hardware platform vendors. For the Linux OS, the HAL compromises of kernel-space drivers.

The IP drivers are generated from the available IP-XACT component descriptions. The IP-XACT component gives the information on IP interfaces and register map. The code 4.1 shows an example IP-XACT description for a register in an IP-XACT component file. For each component IP in an IP-XACT SoC, registers and bit fields mapping can be extracted from their respective IP-XACT component descriptions. This work has only considered memory-mapped HW IPs. However, the principles used for code generation are generic and compliant to the IP-XACT standard, hence are extendible.

As a first step, all the IP-XACT components files from an IP library are parsed to generate a *C struct* of *register map* for each component. The *register map C struct* such prepared has the offset, value, and mask information for each register of an IP component. Here, the *mask* can be set to read or write to bit-field values of a register. The code 4.2 shows an example of *C struct* for *register map* of a sample VDMA (video direct memory access) IP component.

There are additional functions generated to read/write to the registers by individually addressing, or by writing the registers according to the values defined in the *register map C struct*. In case of writing the register values w.r.t. the *register map C struct*, we call it setting an IP component to a specific *mode*. These generated functions are as shown in the code 4.3.

```
<spirit:register>
    <spirit:name>myRegister</spirit:name>
    <spirit:addressOffset>0x04</spirit:addressOffset>
    <spirit:size>32</spirit:size>
    <spirit:field>
        <spirit:name>mySliceA</spirit:name>
        <spirit:bitOffset>0</spirit:bitOffset>
        <spirit:bitWidth>24</spirit:bitWidth>
        <spirit:access>read-only</spirit:access>
    </spirit:field>
     ...
</spirit:register>
```

Code 4.1: Sample IP-XACT description for *register*

```
typedef struct {
    unsigned int offset;
    unsigned int value;
    unsigned int mask;
} RegType;

typedef struct {
  RegType MM2S_DMACR;
  RegType S2MM_DMACR;
  RegType S2MM_DMASR;
  RegType MM2S_START_ADDRESS1;
    //... 55 regs not shown here for simplicity//
} VDMAIP_RegMap;
```

Code 4.2: Sample VDMA *registers struct* generated from IP-XACT component description.

```
void VDMAIP_RegWrite(unsigned int addr, unsigned int mask, unsigned
    int value);

unsigned int VDMAIP_RegRead(unsigned int addr);

static void SetHAMode(VDMAIP_RegMap *mode, unsigned int baseaddr);
```

Code 4.3: Generate IP drivers functions to handle register read-write operations

Various functional operations of an IP component are controlled by writing appropriate values to its control registers. The most typical HW-access tasks include IP initialization (_INIT), availability check (IS_BUSY), start HW processing (_START), stop HW processing (_STOP), control handling among CPU and HW IP, and interrupt handling. These are basically the different operational modes of an IP. The generated IP drivers have functions to configure an IP to a desired functional mode using the *mode struct*. Code 4.4 shows an example *mode struct* and associated mode-set functions for a sample VDMA IP. The associated mode-set functions use the `SetHAMode(<args>)` function to write the control register values. By default, the code generator generates the *mode structs* for HW initialization, start and stop modes (_INIT,_START,_STOP), which can be extended further by any other custom modes.

```
VDMAIP_RegMap __VDMAInitMode = {
.MM2S_DMACR = {.offset=0x00, .mask=0x00000004, .value=0x00000004},
.S2MM_VSIZE = {.offset=0xa0, .mask=0x00000000, .value=0xffffffff},
.S2MM_HSIZE = {.offset=0xa4, .mask=0x00000000, .value=0xffffffff},
.S2MM_START_ADDRESS1 = {.offset=0xac, .mask=0x00000000,
                  .value=0xffffffff},
    //... 55 regs not shown here for simplicity//
};

VDMAIP_RegMap __VDMAStartMode = {...};
VDMAIP_RegMap __VDMAStopMode = {...};

void __VDMA_Driver_initialize(<args>);
void __VDMA_Driver_start(<args>);
void __VDMA_Driver_stop(<args>);
bool __VDMA_Driver_isBusy(<args>);
void __VDMA_Driver_ISR(<args>);
```

Code 4.4: *mode struct* and functions for setting IP operation modes

The IP subsystem drivers utilize the information available from the GRIP IP-integration rules. Using the rules, the SW code generator knows the hardware IPs used in an IP subsystem, and generates the drivers for the corresponding IP subsystem.

The IP subsystem drivers have the *RegMap structs* and *mode structs* to configure the functional modes of IP subsystems. Both of these *C structs* are concatenation of the *C structs* of the HW IP components utilized in an IP subsystem (Code 4.5). Similarly, the generated functions to set the functional modes on an IP subsystem, initialization (_INIT), availability check (IS_BUSY), start HW processing (_START), and stop HW processing (_STOP), are also concatenated w.r.t. utilized IP components (Code 4.6).

In the generic HW-SW interface, the generated HW drivers are made available for accessing HW IP subsystems from SW applications. SW methods in the IP library are extended with a HW-SW interface layer to provide a generic interface for a SW application. A few additional functions and configs are also generated to check for the availability of HW IP subsystem in an SoC design. An application task is assigned for HW processing by calling available HW drivers, otherwise the processing is done in SW. The required *flags and configs* are extracted from the considered IP-XACT designs for the generated HW-access drivers and the HW-SW interface.

```
typedef struct {
  sobelfilterIP_RegMap _sobelfilterIP_RegMap;
  VDMAIP_RegMap _VDMAIP_RegMap;
} Subsystem_Rule1_RegMap;
```

Code 4.5: Concatenation of *RegMap struct* for IP subsystems

```
Subsystem_Rule1_RegMap Subsystem_Rule1_InitMode = {
 ._sobelfilterIP_RegMap = {
  .AP_CTRL = {.offset=0x00, .mask=0x00000000, .value=0xffffffff},
  .GIE     = {.offset=0x04, .mask=0x00000000, .value=0xffffffff},
    ...},
 ._VDMAIP_RegMap = {
  .MM2S_DMACR = {.offset=0x00, .mask=0x00000000, .value=0xffffffff},
  .S2MM_DMACR = {.offset=0x30, .mask=0x00000000, .value=0xffffffff},
    ...}
  };

void Subsystem_Rule1_initialize(Subsystem_Rule1_DriverInstance *
   InstancePtr, Subsystem_Rule1_RegMap Subsystem_InitMode);
void Subsystem_Rule1_start(Subsystem_Rule1_DriverInstance *
   InstancePtr, Subsystem_Rule1_RegMap Subsystem_Rule1_StartMode);
void Subsystem_Rule1_stop(Subsystem_Rule1_DriverInstance *
   InstancePtr, Subsystem_Rule1_RegMap Subsystem_Rule1_StopMode);
bool Subsystem_Rule1_isBusy(Subsystem_Rule1_DriverInstance *
   InstancePtr);
```

Code 4.6: *mode struct* for setting IP subsystem operation mode and generated IP-subsystem drivers header

4.4. Hardware-Accelerated Software Library - Bare Metal

Fig. 4.4 shows the hierarchy of generated HW-access drivers and generic HW-SW interface with the Scheduler. These form the hardware-accelerated software library for the bare-metal package. The structure of IP drivers and IP subsystem drivers functions is identical for both the bare-metal and Linux OS packages. The structure is as discussed in the previous section. The generated IP drivers functions use the *register map struct* and the HAL API drivers to access the registers and bit fields of an IP. Code 4.7 are the register-access read-write functions available as HAL from Xilinx board support package.

For the data flow control, commonly used approaches are either acknowledging an issued interrupt or polling of the completion *status bit* within an IP. The interrupt handling requires initialization of interrupts within an application and execution of the corresponding interrupt service routines (ISR) by an application (Code 4.8). The generated HW drivers also contain interrupt initialization routines. The interrupt initialization drivers expect a handle of system's general interrupt controller (GIC) instance, interrupt port ID extracted from IP-XACT, and a pointer to a default ISR. With this interrupt handing, the method registers the IP's ISR with the processing system. For polling based communication, IS_Busy method is generated for IP subsystems which has functions to poll for IP's busy flag. Fig. 4.4 shows generation of HW-access drivers from IP-XACT metadata and shows interface of a software application to the underneath HW IP control registers. The interrupt handling using functions are generated together with IP and IP subsystem drivers, with ISR functions as empty placeholders. The figure shows the hierarchy of the generated HW-access drivers and HW-SW interface.

```
void Xil_Out32(u32 Addr, u32 Value);
u32 Xil_In32(u32 addr);
```

Code 4.7: Register-access read-write HAL functions for Xilinx hardware platforms

```
void vdmaIP_Driver_intrInitialize(vdmaIP_DriverInstance *InstancePtr
    , IntCntrl_t *InterruptController);
void sobelfilterIP_Driver_intrInitialize(
    sobelfilterIP_DriverInstance *InstancePtr, IntCntrl_t *
    InterruptController);

void Subsystem_Rule1_intrInitialize(Subsystem_Rule1_DriverInstance *
    InstancePtr, IntCntrl_t *InterruptController);
```

Code 4.8: Generated functions for Interrupt handling

The HW drivers for the HASL are generated using the Java-based FreeMarker template engine. The code generator is implemented on the eclipse modeling framework (EMF),

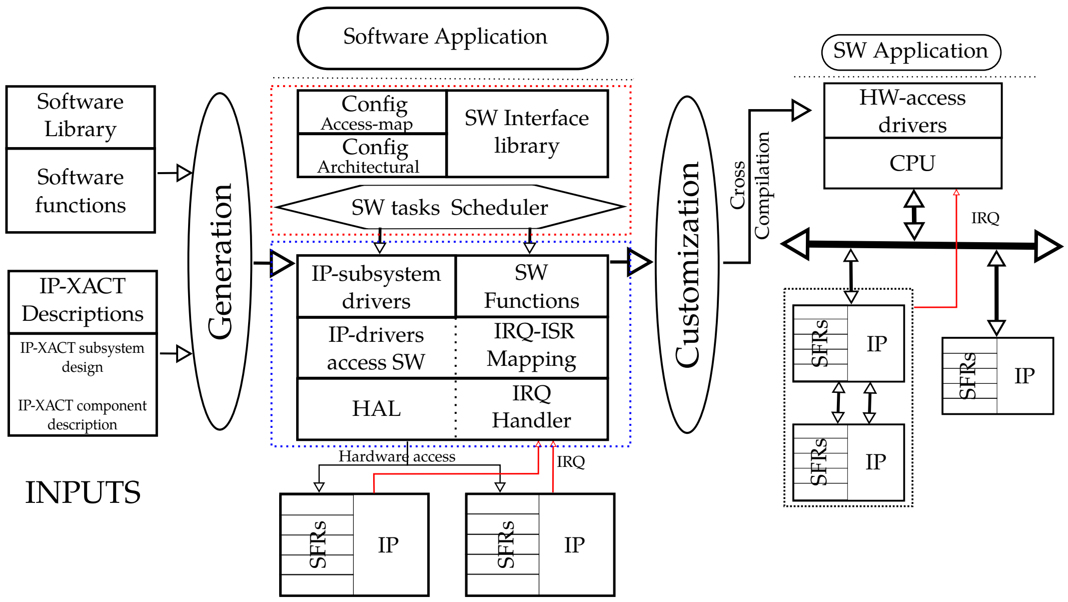


Figure 4.4.: Hierarchy of generated HW drivers for Bare-metal

which uses XML handler to parse IP-XACT files. Fig. 4.5 shows the list of .ftl files as HW drivers templates under their respective generation hierarchy. The FTL templates under `IPsDrivers->IP->drivers` directory are for generating IP drivers for configuring *mode* operations. The templates under `IPsDrivers->IP->platform` are for HAL. The directory `IPSubsysDrivers->Rule->drivers` has templates for generating IP subsystem drivers with concatenated IP drivers. While the directory `IPSubsysIFDrivers->implementation->Rule` has templates for interface functions for the Scheduler, which are later made available to the generic HW-SW interface. Further, Fig. 4.6 shows the HW-access drivers generated for an example IP-integration rule containing two HW IP components - VDMA IP and Sobel filter IP. The generated C/C++ header file of the HW-SW interface can be included in the software applications (for bare-metal projects) to execute generated SW functions, similar to function calls to SW library functions.

4.5. Hardware-Accelerated Software Library - Linux OS

When generating the drivers for the Linux OS, we must consider the details of OS architecture. There are three important elements in the OS that make a SW application access available HW resources: kernel-space drivers, user-space drivers, and device tree blob (DTB). An OS divides the virtual memory into kernel and user spaces. The kernel space is only accessible to privileged kernel operations, while user space memory is available for application software and other HW drivers. The DTB is a compiled binary generated from a device tree source (DTS). The DTS is a tree-based data structure to

```
FTL-Templates
├── IPsDrivers
│   └── IP
│       ├── drivers
│       │   ├── IPS_DRIVERS_driver_C.ftl
│       │   ├── IPS_DRIVERS_driver_H.ftl
│       │   └── IPS_DRIVERS_ipconfig_H.ftl
│       └── platform
│           ├── IPS_PLATFORM_common_C.ftl
│           ├── IPS_PLATFORM_common_H.ftl
│           └── IPS_PLATFORM_platform_config_H.ftl
├── IPSubsysDrivers
│   └── Rule
│       └── drivers
│           ├── IP_SUBSYS_DRIVERS_config_H.ftl
│           ├── IP_SUBSYS_DRIVERS_driver_C.ftl
│           └── IP_SUBSYS_DRIVERS_driver_H.ftl
└── IPSubsysIFDrivers
    └── implementation
        └── Rule
            ├── IP_SUBSYS_IF_DRIVERS_config_H.ftl
            ├── IP_SUBSYS_IF_DRIVERS_function_C.ftl
            └── IP_SUBSYS_IF_DRIVERS_function_H.ftl
```

Figure 4.5.: FreeMarker template files for generation of HW drivers

describe hardware system architectures. Fig. 4.7 shows these three elements, their build steps, and inclusion in the Linux kernel.

There are two key differences of using the generic HW-SW interface in the Linux OS compared to the bare-metal,

- With the Linux OS, the HW-drivers generation is a two step process, generation for – kernel-space drivers and user-space drivers. For the kernel space, it also requires to generate device tree source file. These two drivers spaces are bridged by a set of Linux utility functions. In the Bare-metal implementation, the software application can directly access the available HW resources by using the generated HW drivers.
- In the Linux OS, the hardware abstraction layer (HAL) drivers are handled as kernel-space drivers, which are pre-compiled in the Linux kernel and are accessed from the software application using the generated user-space drivers. For the bare-metal, all HW-access drivers are complied together with the software application.

The Linux OS prevents a software application to directly access the SoC hardware resources. When building the Linux kernel, the available HW subsystems of the SoC must be defined in the device tree source (DTS) file, which is included in the Linux boot-up

```
IPSubsysIFDrivers
└─ implementation
   ├─ Rule1
   │  ├─ config.h
   │  ├─ function.c
   │  └─ function.h
   └─ Rule2
      └─ ⋮
IPSubsysDrivers
└─ Rule1
   ├─ drivers
   │  ├─ config.h
   │  ├─ driver.c
   │  └─ driver.h
   └─ Rule2
      └─ ⋮

IPsDrivers
└─ vdmaIP
   ├─ drivers
   │  ├─ ipconfig.h
   │  ├─ vdmaIP_driver.c
   │  └─ vdmaIP_driver.h
   ├─ platform
   │  ├─ common.c
   │  ├─ common.h
   │  └─ platform_config.h
   └─ software
      ├─ -
      └─ -

IPsDrivers
└─ sobelIP
   ├─ drivers
   │  ├─ ipconfig.h
   │  ├─ sobelIP_driver.c
   │  └─ sobelIP_driver.h
   ├─ platform
   │  ├─ common.c
   │  ├─ common.h
   │  └─ platform_config.h
   └─ software
      ├─ sobel_sw.c
      └─ sobel_sw.h
```

Figure 4.6.: The generated HW drivers files for bare-metal HASL package

routine. The Linux OS interprets the hardware abstraction information from the DTS to generate system functions (`udev` or HAL). The DTS is targeted to a desired HW platform. In this work, the GRIP tool generates the DTS file for the custom HW IPs in the programmable logic (PL) part of Xilinx Zynq chipset (Xilinx 2016). The code 4.9 shows a new device node for a custom HW IP, `sobel_filter_top`, in the generated DTS file. This DTS file is generated using the IP-XACT SoC descriptions.

```
        sobel: sobel_filter_top@71800000 {
                compatible = "generic-uio";
                reg = < 0x71800000 0x10000 >;
                interrupts = < 0 35 0 >;
                interrupt-parent = <&gic>;
        };
```

Code 4.9: An example of a generated device node in a DTS fils

The kernel-space drivers consist of functions with pre-defined interfaces, which are accessible to the user space using only the system calls. Depending on the memory accessibility of HW devices, the kernel drivers can be of following types: character device drivers, network device drivers, block device drivers etc. Since in this work the acceleration of software functions using memory-mapped hardware accelerators is of interest, only the character device drivers are of interest. For this, the kernel drivers from user-space I/O drivers (UIO) provide sufficient functions to access memory-mapped address space of the HW accelerators used in the context of this work (Corbet et al. 2005). First, the available devices are recognized in the DTS, then the recognized devices are linked

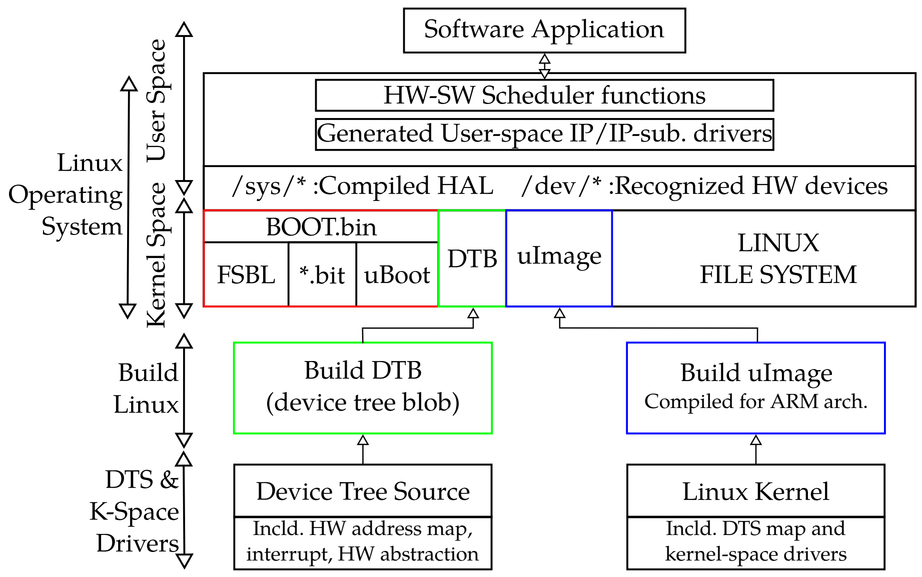


Figure 4.7.: Essential elements of the Linux kernel facilitating access HW devices

to the corresponding kernel-space functions using the `compatible = "generic-uio"` string (Code 4.9). The kernel-space C struct in the Code 4.10 declares the system call mappings to the kernel-space functions. In this case, `.write` will be the system call to access `uio_write` function in the kernel space to write to HW device registers.

The user-space drivers are generated from IP packages as discussed in Section 4.3. Figure 4.8 shows the hierarchy of the generated HW drivers for the Linux OS. For Linux OS, `udev` is a device manager, which manages the recognized device by the Linux kernel. For each recognized driver, a device node is created in the `\dev` directory. The user-space drivers can access the recognized devices using system calls.

```
static const struct file_operations uio_fops = {
        .owner          = THIS_MODULE,
        .open           = uio_open,
        .release        = uio_release,
        .read           = uio_read,
        .write          = uio_write,
        .mmap           = uio_mmap,
        .poll           = uio_poll,
        .fasync         = uio_fasync,
        .llseek         = noop_llseek,
};
```

Code 4.10: Kernel-space drivers mapping to corresponding system calls in the user space

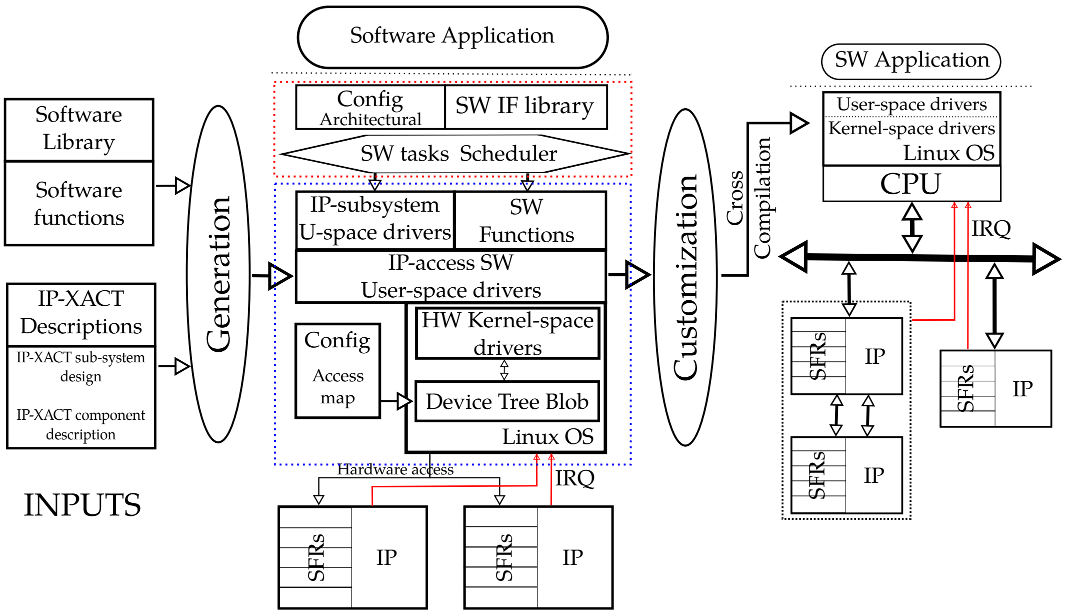


Figure 4.8.: Hierarchy of generated HW drivers for Linux OS

For the Linux OS, in addition to previously discussed HW drivers, it is also required that a few additional functions for linking the recognized devices (HW subsystems) to the user-space drivers are defined. The user-space drivers are appended by a set of utility functions to access the hardware devices from the `/dev/` and `/sys/` directories under the Linux file system. These utilities are implemented using some available Unix binary functions. The code 4.11 shows the Linux utility functions. These functions are used in the user space together with the generated user-space drivers to recognize the HW accelerating sub-systems in the SoC.

```
void resetDev(); // Resets the Sys devices

int HAexists(const char *archDir, const char *haName,
        int level, const char *fil); // Returns # of HA of same type

char *getDev(const char *archDir, const char *haName,
        int level, const char *fil); // Returns the device node name
 };
```

Code 4.11: The Linux utility functions in the user space to access the HW devices

4.6. Software Application Development using HASL

In this section, we will look at using the hardware-accelerated software library from the software developers' view. Fig. 4.9 shows the containment of the HASL. A software developer can use HASL in place of a SW library, in order to efficiently utilize available HW resources in a candidate SoC. The software application calls HW-accelerated SW functions, which call their scheduler.

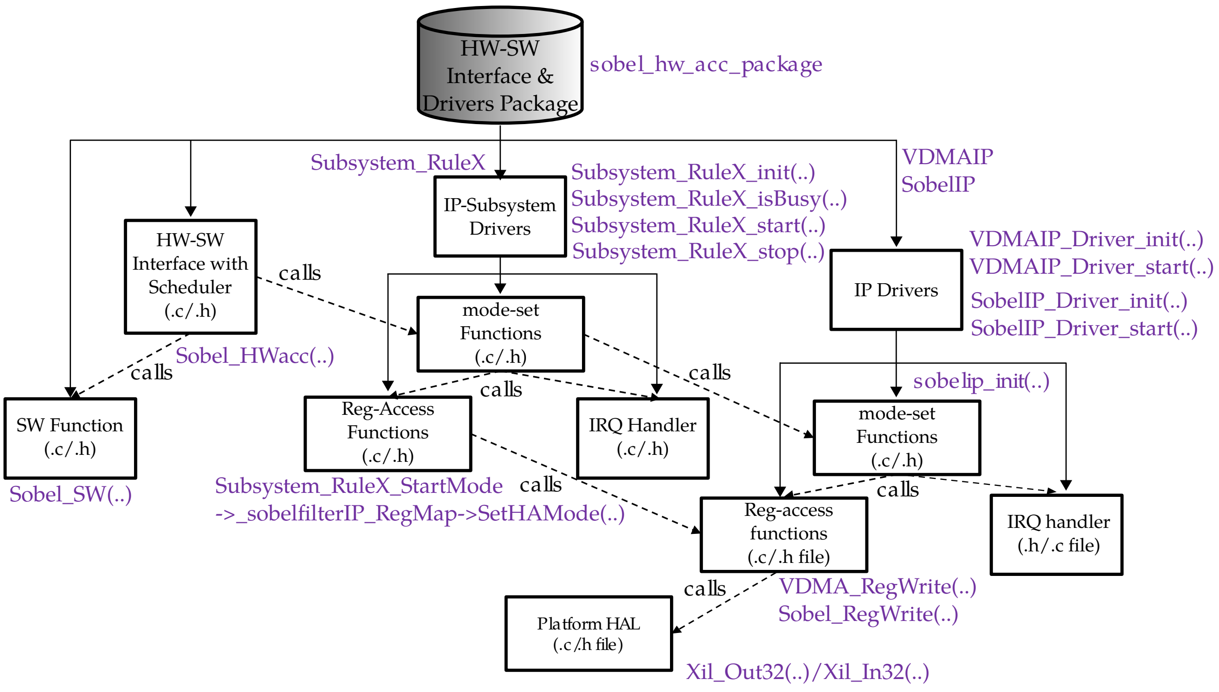


Figure 4.9.: Generated hardware drivers package containment.

Fig. 4.9 illustrates the hierarchy of function calls between the generated software codes. It shows the hierarchical calls for an example IP subsystem with a VDMA (video direct memory access) and a Sobel-filter IP. The generated functions package, `sobel_hw_acc_package`, corresponds to one specific IP-integration rule for this IP subsystem. The SW can call the generic HW-SW interface function (`Sobel_HWacc()`). This call goes to either the `Sobel_SW(..)` function, which is the SW implementation of the Sobel filtering task, or it calls the HW-access drivers. The IP subsystem functions (`Subsystem_RuleX_*()`) to configure the IP subsystem access the required IP components as described in the corresponding IP-integration rules. The IP subsystem drivers utilize the generated HW drivers of individual IPs of the IP library (`VDMAIP_Driver_*(), SobelIP_Driver_*()`). Further, the HAL uses HW platform specific drivers (`Xil_Out32(..)/Xil_In32(..)`) to read and write to the IP registers and bit fields. These generated HW drivers together with the hardware information that comes with the IP packages make the complete HASL. In the HASL, the HW drivers are still configurable as the final SoC architecture is not yet generated. The generation of mode structs and driver code enables the IP supplier to quickly generate major parts of the driver layer of the HASL. The IP supplier can customize the driver layer further as required by the IPs, while configuring the control register values.

4.7. Conclusions

In this chapter, we discussed the challenges of adapting the software application for the SoC architecture changes. For complex SoC designs, the software developers lack sufficient HW knowledge to efficiently utilize the available HW resources for optimizing the application performances. We extended the available HW knowledge in the proposed IP packaging (Chapter 3) to generate the HW drivers, which comprises HW-access drivers and a generic HW-SW scheduler. These HW drivers are generated using IP-XACT descriptions of the available IPs and the IP-integration rules.

The domain-specific IP packages and the associated HW drivers form the hardware-accelerated software library (HASL). In the next chapter, we will utilize the HASL to perform IP-integration and SoC design space exploration.

5. Automated IP-Integration and Design Space Exploration of SoCs

"While the goal of purchasing IP is to reduce the engineering effort, the process of integrating IP is not a "drop and done" process — it requires a lot of engineering skills and effort for it to be successful.", Helena Zheng, VP, 3DSP Corp.

5.1. Introduction and Problem

In the previous two chapters, we discussed the IP-packaging and hardware-accelerated IP library (HASL) preparation steps from the perspective of an IP supplier. This chapter will look at the GRIP tool utilities for an SoC developer. With the existing IP-integration methodologies, the SoC architect faces the following challenges:

1. The SoC architect is required to understand the details of HW interfaces of IP components, before they can be correctly integrated in an SoC. After introducing structural changes in an SoC, the new candidate SoC must be validated for correctness and design integrity for the HW synthesis. This is the problem of system integrity and HW-SW bring up. The challenges originate because of the SoC architect's lack of sufficient knowledge on the third-party IPs.
2. In SoC designs, typically, multiple targeted design abstraction levels, e.g. TLM in SystemC, RTL on an FPGA, are desired for an SoC at various stages of the SoC design flow. It requires a laborious effort to implement HW-SW system models for all these abstraction levels.
3. For the target-platform code generation, the key challenges are three fold: a) mapping the IP-XACT SoC descriptions to the synthesizable HW descriptions of the target platform; b) integrating the HW drivers with the SW application (bare metal or with OS) and the SW cross-compilation; c) enabling a seamless interface between the SW and HW resources via the interface routines (Scheduler).

This chapter will discuss the algorithms to automate IP integration and enable SoC design space exploration using HASL. The automated IP integration and HW-SW bring-up on a targeted HW platform requires seamless alignment of the following aspects of a HW-SW system: generating a synthesizable SoC, generating HW drivers, and adapting the SW application to utilize HW resources. Further, we will also discuss the integration of the GRIP tool with the external HW synthesis tools and building a custom Linux OS for the generated candidate SoCs.

This chapter discusses the following three contributions of this work,

1. The GRIP IP-integration engine to automate the process of IP integration for SoC design. We will discuss the utilized automation algorithms and the model-based implementation of the engine.
2. The GRIP design space exploration (DSE) engine to explore the SoC design space for a software application targeted on the HASL. This SoC design space is encoded by the IP-integration rules of a domain-specific HASL.
3. The GRIP code generation engine for generating HW and SW projects targeted for Xilinx FPGA prototyping. We will discuss the transfer of SoC design descriptions from the GRIP environment to the Xilinx design flow.

5.2. Model-Based Graph Rewriting

In Chapter 1, we discussed the background on the mathematics of graph grammars to describe structural systems and grammar rules to describe structural changes on graphs. In this chapter, we will build on that foundation to describe model-based mathematics and algorithms utilized in this work to perform structural transformations. In the following, we describe the model-based process of graph rewriting. Our approach is general in the sense that there is no restriction on the models as long as their relation can be expressed in a graph. In this section we give the general description of the model-based graph rewriting engine. Its application on IP integration is illustrated in Section 5.3. To start, we need a host graph and rewriting rule defined as follows:

Host graph: The input graph $G = (N_G, E_G)$ to the graph rewriting process is called the host graph. In order to support model-driven principles, meta models are defined for possibly multiple node and edge types. In our approach the EMF Ecore meta-modeling framework is used to define node and edge classes.

Graph Rewriting Rule: We define a graph rewriting rule $\mathfrak{R}_i$ as a triple, $\mathfrak{R}_i = (L_i, R_i, m_{R2L,i})$. L_i and R_i are two connected graphs. $m_{R2L,i}$ is a bijection between the two graphs.

LHS pattern graph: $L_i = (N_{L,i}, E_{L,i})$ is called the left-hand-side (LHS) pattern graph of the rule. For every node l_x and edge (l_y, l_z), with $l_x \in N_{L,i}, (l_y, l_z) \in E_{L,i}$, we define a node-level match function `matchnode` and edge-level match function `matchedge`. The `matchnode` function takes as input one LHS node l_x and one host graph node $n_a \in N_G$. It evaluates to true, when the properties of n_a satisfy the match pattern given in the pattern node l_x of the LHS, otherwise it evaluates to false. The same applies for the edge patterns. We use Epsilon validation language (EVL) to implement the match patterns for the LHS nodes and LHS edges. As already described in Sec. 2.2.5, Epsilon provides a family of languages and tools for model operations and EVL is a model validation language of the Epsilon family. The match functions just need to evaluate the EVL script on the host graph node because its class is based on an Ecore meta-model.

RHS rewriting graph: $R_i = (N_{M,i} \cup N_{C,i}, E_{M,i} \cup E_{C,i})$ is called the right-hand-side (RHS) rewriting graph of the rule $\mathfrak{R}_i$ with two different node and edge sets. The node set $N_{M,i}$ and edge set $E_{M,i}$ of the RHS are modify nodes and edges with associated modify functions `modifynode` and `modifyedge`. The modify functions `modifynode` and `modifyedge` take as input a host graph node or edge and return a new host graph node or edge. In contrast, the create functions `createnode` and `createedge` of the create node set $N_{C,i}$ and edge set $E_{C,i}$ take no input but also return a new host graph node or edge. We use EOL scripts to implement the modify and create functions.

Mapping for Modification Nodes: For the modification nodes, an additional bijection $m_{R2L,i} : N_{M,i} \rightarrow \hat{N}_{L_i} \subseteq N_{L,i}, E_{M,i} \rightarrow \hat{E}_{L,i} \subseteq E_{L,i}$ between LHS and RHS is given. It is used to determine the nodes in the host graph that need to be modified.

Rule Execution Steps: Given a set of rules R_i and a host graph G, graph rewriting is a three step process:

1. Match: In the match step, all possible graph matches $\hat{G}_{i,j}$ of the LHS graphs L_i for all rules $\mathfrak{R}_i$ are found in the host graph G with their respective match function $m_{L2G,i,j}$.
2. Choose: One match $\hat{G}_{i,j}$ of L_i in G is chosen.
3. Apply: The RHS R_i is applied to the chosen match $\hat{G}_{i,j}$ based on the respective matching function $m_{L2G,i,j}$ for rewriting graph G to attain a new graph F.

Rule Match step: The first step of the graph rewriting is the matching. Given an host graph $G = (N_G, E_G)$ and the left-hand-side graph $L_i = (N_{L,i}, E_{L,i})$ of a rule $\mathfrak{R}_i$, the subgraph $\hat{G}_{i,j}$ of G is called a match of L_i in G if:

- $\hat{G}_{i,j} = (\hat{N}_{G,(i,j)}, \hat{E}_{G,(i,j)})$ is isomorphic to L_i w.r.t. the bijection $m_{L2G,i,j} : N_{L,i} \rightarrow \hat{N}_{G,(i,j)} \subseteq N_G, E_{L,i} \rightarrow \hat{E}_{G,(i,j)} \subseteq E_G$.
- $m_{L2G,i,j}$ is called the matching function and must be total.
- Foreach $n_a \in \hat{N}_{G,(i,j)}$ with $m_{L2G,i,j}(n_a) = l_x$ it must hold:
  `matchnode` $(l_x, n_a) = \text{True}$
- Foreach $(n_b, n_c) \in \hat{E}_{G,(i,j)}$ with $m_{L2G,i,j}((n_b, n_c)) = (l_y, l_z)$ it must hold:
  `matchedge` $((l_y, l_z), (n_b, n_c)) = \text{True}$

Basically, we look for a subgraph in G, which has the same node-edge structure as the L_i and whose nodes' and edges' properties match the patterns given in the LHS nodes and edges. There can be any number (including zero) of matches $\hat{G}_{i,j}$ of L_i in G.

Algorithm 1 illustrates our implementation of the Matching algorithm. The matching function returns a set, $\mathcal{M}_{L2G,i}$, of all the complete matches of L_i in a host graph G. The set $\mathcal{M}_{L2G,i} = \{m_{L2G,i,j}\}_{j\in\{1,2,\ldots,n\}}$ is a collection of n matching functions from L_i to G for n matches. In the algorithm, at first, a root node of L_i is randomly picked and matched to a node in the G (if a match is possible). Then, iteratively the matching is expanded to nodes connected to the root node of L_i. With each iteration, the matching function continues to append the matching function $m_{L2G,i,j}$ with new mappings. Further, the matching continues from the matched nodes of L_i to unmatched nodes, till all nodes of L_i are explored. The algorithm removes a match function $m_{L2G,i,j}$ from $\mathcal{M}_{L2G,i}$, when a node $l_x \in L_i$ does not find a match in G (i.e. $m_{L2G,i,j}(l_x) = null$).

Rule Choose Step: There can be several rules with several matches for G. In the choose step, one rule $\mathfrak{R}_i$ and one respective match $\hat{G}_{i,j}$ (by default, the first complete match for fast tree exploration) are selected for application.

Rule Application Step: The application will yield a new final graph F by rewriting the matched subgraph $\hat{G}_{i,j}$ in G with the RHS graph R_i. Our implementation of the rule application step is shown in Algorithm 2. At first, corresponding to all the matched nodes and edges between L_i and R_i, modified nodes and edges are created in the final graph F using `modifynode` and `modifyedge` functions. Similarly, by using `createnode` and `createedge` corresponding to unmatched nodes and edges among L_i and R_i new nodes and edges are created in the F. Finally, the unmatched nodes and edges among L_i and G are created in F. This yields a new final graph F from a host graph G after applying rule R_i.

5.3. Automated IP Integration

In order to utilize the mathematics of graph grammars for automating SoC IP integration, all the SoC structural changes are performed in the graph space. For this, the current candidate design and the IP-XACT integration rule are first transformed to graph-based representations before the rule application. There are three types of graphs used in this methodology, namely, architectural graphs (AGs), pattern graphs (PGs) and rewriting graphs (RGs). Each AG describes one complete SoC design. PGs define the LHS graph of a rule and RGs are used to describe the RHS graphs. The transformation of IP-XACT SoC design to the corresponding AG has been discussed in Section 3.2.4. In the following, we will discuss about LHS PG and RHS RG generation in detail.

Algorithm 1: Rule Matching Algorithm

```
Input  : Host Graph $G = (N_G, E_G)$, LHS $L_i = (N_{L,i}, E_{L,i})$
Output: Set of matching functions $\mathcal{M}_{L2G,i}$
Match($G$,$L_i$)
 Select $l_{root} \in N_{L,i}$                       // Select a root node from the LHS
 $U_{L,i} \leftarrow N_{L,i} \setminus \{l_{root}\}$                 // $U_{L,i}$ is the set of unmatched nodes
 foreach $n_k \in N_G$ do
    if matchnode $(l_{root}, n_k)$= $True$ then      // Match the LHS root node to the host graph nodes
        $m_{L2G,i,new}(l_{root}) \leftarrow n_k$
        foreach $l_z \in N_{L,i} \setminus \{l_{root}\}$ do
            $m_{L2G,i,new}(l_z) \leftarrow null$
        end
        $\mathcal{M}_{L2G,i} \leftarrow \mathcal{M}_{L2G,i} \cup \{m_{L2G,i,new}\}$       // Append all matches in the host graph to $\mathcal{M}_{L2G,i}$
    end
    while $U_{L,i} \neq \emptyset$ do
        $V \leftarrow \emptyset$
        foreach $(l_x, l_y) \in E_{L,i}, l_x \in U_{L,i}, l_y \in N_{L,i} \setminus U_{L,i}$ do  // $(l_x, l_y)$ Expand the matched sub-
            $\mathcal{M}_{L2G,i} \leftarrow$ mapNextNode($l_x, l_y, \mathcal{M}_{L2G,i}, G$)         // graph to the unmatched nodes of $L_i$
            $V \leftarrow V \cup \{l_x\}$
        end
        $U_{L,i} \leftarrow U_{L,i} \setminus V$                          // Remove matched nodes from $U_{L,i}$
    end
    return $\mathcal{M}_{L2G,i}$
end
mapNextNode($l_x, l_y, \mathcal{M}_{L2G,i}$, $G$)                   // Match $l_x \in L_i$ to a node of host graph $G$
 $\mathcal{M}_{L2G,new} \leftarrow \emptyset$
 foreach $m_{L2G,i,j} \in \mathcal{M}_{L2G,i}$ do
    foreach $n_k \in N_G$ do
        if matchnode($l_x, n_k$) & matchedge($(l_x, l_y), (n_k, m_{L2G,i,j}(l_y))$) & $\nexists_{l_z \in N_{L,i}} m_{L2G,i,j}(l_z) = n_k$
         then
            if $m_{L2G,i,j}(l_x) = null$ then           // If match is not set for $l_x$, new $m_{L2G,i,j}$ is set
                $m_{L2G,i,j}(l_x) \leftarrow n_k$
                $m_{L2G,i,j}(l_x, l_y) \leftarrow (n_k, m_{L2G,i,j}(l_y))$
            end
            else        // If match already exists for $l_x$, create $m_{L2G,i,new}$ and append it to $\mathcal{M}_{L2G,i}$
                $m_{L2G,i,new} \leftarrow m_{L2G,i,j}$
                $m_{L2G,i,new}(l_x) \leftarrow n_k$
                $m_{L2G,i,new}(l_x, l_y) \leftarrow (n_k, m_{L2G,i,j}(l_y))$
                $\mathcal{M}_{L2G,new} \leftarrow \mathcal{M}_{L2G,new} \cup \{m_{L2G,i,new}\}$
            end
        end
    end
end
foreach $m_{L2G,i,j} \in \mathcal{M}_{L2G,i}$ do             // Remove partial matches from $\mathcal{M}_{L2G,i}$
    if $m_{L2G,i,j}(l_x) = null$ then
        $\mathcal{M}_{L2G,i} \leftarrow \mathcal{M}_{L2G,i} \setminus \{m_{L2G,i,j}\}$
    end
end
return $\mathcal{M}_{L2G,i} \cup \mathcal{M}_{L2G,new}$
```

Algorithm 2: Rule Application Algorithm

```
Input  : Host Graph $G = (N_G, E_G)$, RHS $R_i = (N_{M,i} \cup N_{C,i}, E_{M,i} \cup E_{C,i})$, LHS $L_i = (N_{L,i}, E_{L,i})$,
         Mapping $m_{R2L,i}$, Match function $m_{L2G,i,j}$
Output: Rewritten Graph $F = (N_f, E_f)$
ApplyRule($G, L_i, R_i, m_{R2L,i}, m_{L2G,i,j}$)
foreach $r_s \in (N_{C,i} \cup N_{M,i})$ do          // Initialize create/modify nodes $N_{C,i}/N_{M,i}$ of $R_i$
    $m_{R2F}(r_s) \leftarrow null$                     // Reset match function $m_{R2F}$ from $R_i$ to $F$
end
foreach $n_g \in N_g$ do                              // Reset match function $m_{G2F}$ from $G$ to $F$
    $m_{G2F}(n_g) \leftarrow null$
end
foreach $r_c \in N_{C,i}$ do                          // Create new nodes in $N_f$ corresponding to $N_{C,i}$
    $n_f \leftarrow$ createnode($r_c$)
    $N_f \leftarrow N_f \cup \{n_f\}$
    $m_{R2F}(r_c) \leftarrow n_f$
end
foreach $r_m \in N_{M,i}$ do                          // Create modified nodes in $N_f$ corresponding to $N_{M,i}$
    $n_g \leftarrow m_{L2G,i,j}(m_{R2L,i}(r_m))$
    $n_f \leftarrow$ modifynode($r_m, n_g$)
    $N_f \leftarrow N_f \cup \{n_f\}$
    $m_{R2F}(r_m) \leftarrow n_f$
    $m_{G2F}(n_g) \leftarrow n_f$
end
foreach $(r_x, r_y) \in E_{C,i}$ do                   // Create new edges in $E_f$ corresponding to $E_{C,i}$
    $(m_{R2F}(r_x), m_{R2F}(r_y)) \leftarrow$ createedge($(r_x, r_y)$)
    $E_f = E_f \cup \{(m_{R2F}(r_x), m_{R2F}(r_y))\}$
end
foreach $(r_x, r_y) \in E_{M,i}$ do                   // Create modified edges in $E_f$ corresponding to $E_{M,i}$
    $(n_g, n_h) \leftarrow m_{L2G,i,j}(m_{R2L,i}((r_x, r_y)))$
    $(m_{R2F}(r_x), m_{R2F}(r_y)) \leftarrow$ modifyedge($(r_x, r_y), (n_g, n_h)$)
    $E_f \leftarrow E_f \cup (m_{R2F}(r_x), m_{R2F}(r_y))$
end
foreach $n_g \in N_G$ do                              // Create unmatched nodes ($L_i$ matching with $G$) of $N_G$ in $N_f$
    if $\nexists_{l_x \in N_L} m_{i,j}(l_x) = n_g$ then
        $n_f \leftarrow n_g$
        $N_f \leftarrow N_f \cup \{n_f\}$
        $m_{G2F}(n_g) \leftarrow n_f$
    end
end
foreach $(n_g, n_h) \in E_G$ do                       // Create unmatched edges ($L_i$ matching with $G$) of $E_G$ in $E_f$
    if $\nexists_{(l_x, l_y) \in E_l} (m_{i,j}(l_x) = n_g \;\&\; m_{L2G,i,j}(l_y) = n_h)$ then
        if $m_{G2F}(n_g) \neq null \;\&\; m_{G2F}(n_h) \neq null$ then
            $(m_{G2F}(n_g), m_{G2F}(n_h)) \leftarrow (n_g, n_h)$
            $E_f \leftarrow E_f \cup \{(m_{G2F}(n_g), m_{G2F}(n_h))\}$
        end
    end
end
```

5.3.1. LHS Generation

Each IP-integration rule in the HASL consists of one LHS and one RHS IP-XACT design object. The LHS pattern graph is generated from IP-XACT in the IP-XACT2PG transformation using only the LHS IP-XACT design object. From the component instances all set properties are translated into EVL constraints except the instance name. This includes VLNV identifier and all set configuration values. The EVL constraints are evaluated with the matching functions `matchnode` and `matchedge` as was described in Section 5.2.

5.3.2. RHS Generation

The RHS rewriting graph is generated in the IP-XACT2RG transformation, which requires to analyze the LHS and RHS IP-XACT designs. For each component of the RHS design object, it is first checked if there exists a component with the same instance name in the LHS design object. If this is the case, a modify node is created in the RHS rewriting graph, otherwise a create node. The rewriting operations are described in the EOL model language from the Epsilon project. EOL scripts allow to modify and generate new model instances based on the Ecore meta-model used for AG nodes and AG edges. The modify node describes the differences in the properties of the LHS and RHS. All differences are implemented as modifications in the EOL script of the node called by the `modifynode` and `modifyedge` functions. Properties which are not set in the LHS and RHS IP-XACT components are left untouched during modification. This allows e.g. to define a context node to set conditions for matching, which would have an empty EOL modification script. The create nodes and edges copy all properties set in the RHS component to a EOL script to create a new node or edge with the `createnode` and `createedge` functions.

5.3.3. IP Integration

The inputs to the GRIP tool are: a) an input IP-XACT host SoC design, b) an IP-XACT IP-integration rule, and c) input software application; and the final outputs are: a) a new IP-XACT SoC design with the changes described by the integration rule, and b) HW drivers for the SW application. The output IP-XACT design can be taken further through GRIP code generation to generate hardware description files for a target hardware platform, e.g. an FPGA prototyping board. Fig. 5.1 describes the flow diagram for IP-integration using GRIP. The key component doing the structural transformations is the *Graph-Rewriting Engine* using the algorithms as described in Section 5.2. Another key component is the OCL-based *IP-XACT Verification Engine*. The verification engine performs the design verification checks on the given IP-XACT design description. The verification engine complies to a predefined set of design rule checks described using

OCL scripts, which check for the correctness of bus interconnections and protocols, and signal ports integrity of the design connections w.r.t. to IP components in the IP-library, including clocks and interrupts. It is an important step to assure that the ports conflicts and the component instances conflicts are correctly handled by the GRIP engine to keep the design integrity during the structural transformations.

Additionally, in the GRIP tool, there are multiple model-to-model (M2M) transformation engines, which perform the required transformations from the IP-XACT space to the graph space and vice-versa. In Fig. 5.1, as a first step, an input SoC design goes through the verification checks, and then M2M transformation to generate an initial design graph. The second input, an IP-integration rule, has LHS and RHS IP-XACT designs. The LHS design goes through M2M transformation to generate an EVL Pattern graph, while the RHS design transforms to generate an EOL rewrite graph. The RHS design transformation also requires the LHS pattern design and initial host SoC design to generate EOL rewrite graph. All these graph inputs go to the graph grammar engine, which generates the new design graph.

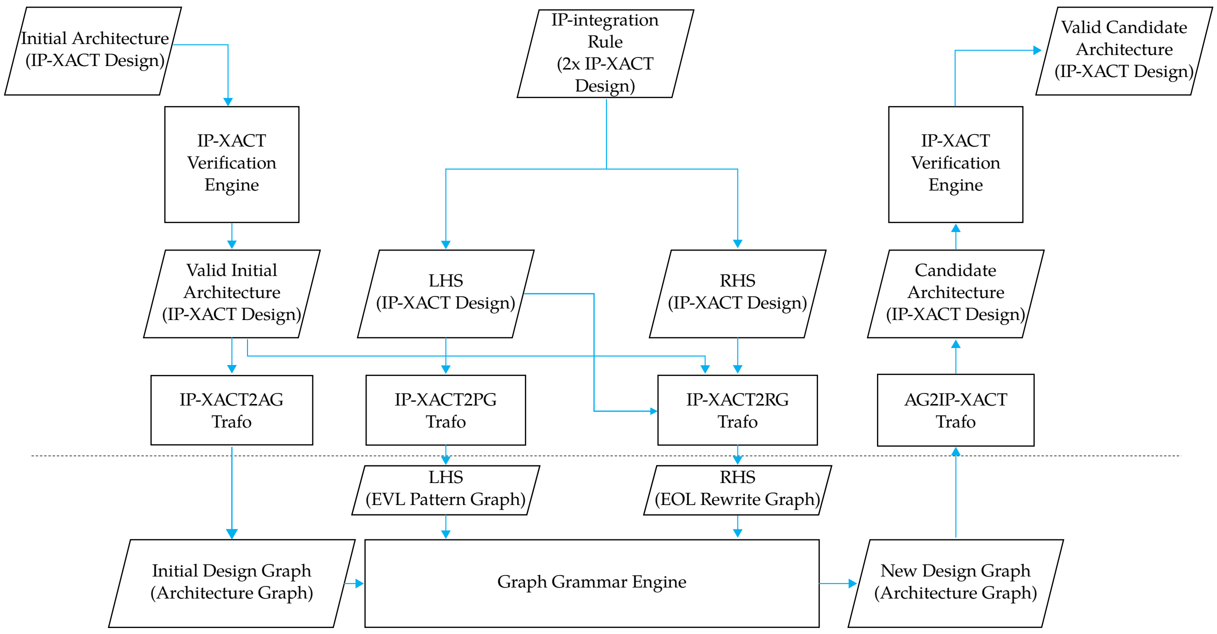

Initial Architecture (IP-XACT Design)
IP-integration Rule (2x IP-XACT Design)
Valid Candidate Architecture (IP-XACT Design)
IP-XACT Verification Engine
IP-XACT Verification Engine
Valid Initial Architecture (IP-XACT Design)
LHS (IP-XACT Design)
RHS (IP-XACT Design)
Candidate Architecture (IP-XACT Design)
IP-XACT2AG Trafo
IP-XACT2PG Trafo
IP-XACT2RG Trafo
AG2IP-XACT Trafo
LHS (EVL Pattern Graph)
RHS (EOL Rewrite Graph)
Initial Design Graph (Architecture Graph)
Graph Grammar Engine
New Design Graph (Architecture Graph)

Figure 5.1.: Flow diagram for automated IP-integration using IP-integration rules

Fig. 5.2 describes the flow steps for the *Graph Grammar Engine*. It follows the three graph-rewriting steps, *I. Match, II. Choose, III. Apply*. The matching step searches sub-graphs of the input architectural graph that satisfies LHS. Each EVL pattern graph node or edge stores an EVL script, with which an evaluation function can be executed to check whether the architectural graph node or edge satisfies constraints. This function is used to overload comparison between nodes and edges, therefore this matching problem can be formulated as a standard sub-graph isomorphism problem and can be solved with our own graph isomorphism algorithm or VF2 algorithm in (Cordella et al. 2001). From the graph matcher, multiple matches might be found. Each of these matches records not only the matched sub-graph, but also a mapping function from the matched LHS graph nodes and edges to the architectural graph nodes and edges. A graph can only be rewritten according to one match at a time, so all matches are sent to the selector. In the selector, three selection mode are supported, first-match selection, multiple selection and user defined selection. The first-match selection mode is the fastest, since it selects the first detected match. Multiple selection means all matches are selected and they will be executed one by one and generate multiple separate outputs. In the user-defined selection, the user sets the selection manually. On the selected match, the RHS can be applied to the input architectural graph to generate a new output graph. The RHS operations are implemented by execution of EOL scripts as discussed in Sec. 5.3.2. For modification, the corresponding node or edge in input architectural graph is first cloned, then modified and added to the output graph, so that the output graph contains no reference and is completely independent from the input graph. For the matched LHS nodes in the host graph that do not have an equivalent RHS nodes in the corresponding rule (i.e. deletion of nodes is intended in the rule), the deletion of those matched nodes of the host graph is required. During deletion, all connected edges are deleted as well because it is illegal to have floating edges in graphs. After generating the matched sub-graph, the remaining nodes and edges of the input architecture graph are cloned and added to the output graph as well.

The generated new design graph is then transformed to a candidate IP-XACT SoC design and it goes through the verification checks. After passing the verification checks, the valid candidate architecture is generated.

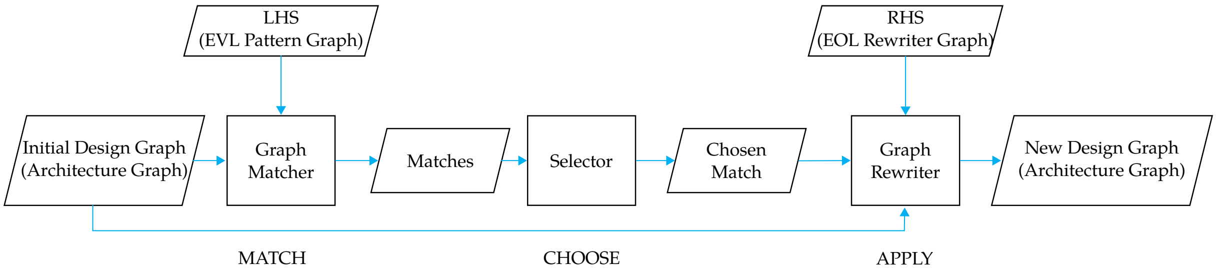


Figure 5.2.: Flow diagram for the GRIP IP-integration Engine.

Fig. 5.3 shows an example of using the GRIP tool for automating IP-integration on a sample input design using the IP-integration Rule-3 in Fig. 3.3. The integration-rule describes the integration of an IP subsystem (VDMA and ER_top) on a dedicated bus-system. The input IP-XACT design is transformed to an architecture graph (the figure shows a simplified AG) to apply the IP-integration rule to first generate the new design graph and later its transformation to the final IP-XACT candidate architecture. In the figure, the blue color nodes and components denote the newly added entities, and the yellow color nodes donates the modified entities from the input design.

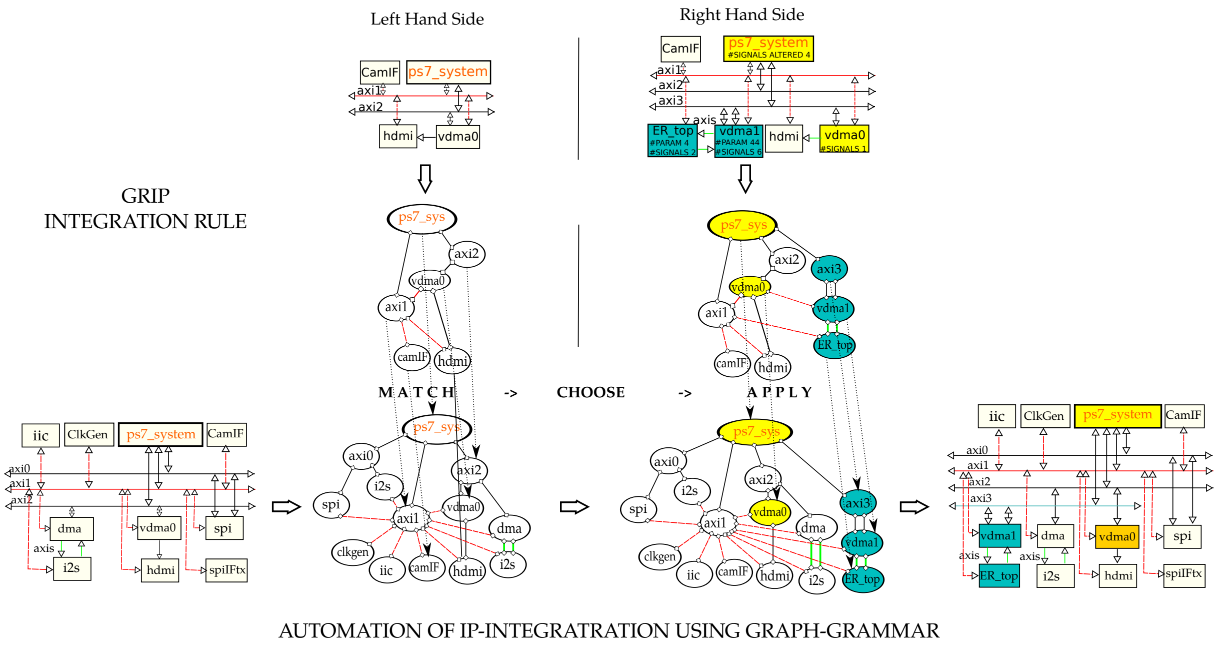


Figure 5.3.: Overview of steps for automated IP-integration in an SoC using the GRIP

5.4. GRIP Design Space Exploration

In the previous section, we mentioned that the graph matcher can find multiple matches of the LHS in the input host design. This leads to the branching and the generation of multiple candidate architectures. In this section, we will discuss the branching of the generated SoCs from the host SoC design.

During SoC design space exploration, an SoC architect iterates over available HW IPs to explore candidate SoC architectures which give best trade-offs for SW application performances and HW costs. A DSE targeted on a domain-specific HW IP library confines the design space to an attainable set of SoCs by that library. The design space exploration starts from an initial SoC design, also called a host SoC. The host SoC contains minimum required HW components to execute the target software applications. These components can be a master CPU, memories, I/O interfaces, and system buses. Then, progressively some of the software tasks are off-loaded from the master CPU to other computing IP subsystem units. The SoC architect must integrate an IP subsystem in desired configurations, and quickly synthesize and bring-up the HW-SW system for application profiling. Another dimension of complexity is various configurations in which a HW IP can be integrated in an SoC, eg. pipeline, parallel, and how the data communication is handled, eg. CPU, or direct memory access (DMA) based, interrupt-based or polling.

In the rest of this section, we will discuss the automation of SoC DSE by iterative usage of GRIP IP-integration engine to explore the SoC design space.

5.4.1. Design Space Exploration (DSE) Tree

In the proposed domain-specific HASL with each IP-integration rule describing a feasible structural change, all the available IP-integration rules together describe all feasible changes on a host SoC. The iterative application of the rules leads to a design space exploration (DSE) tree based on the HASL. In the DSE tree, each node represents a candidate SoC architecture, and each edge is a rule application. The available IP-integration rules in the HASL define the structure of the DSE tree, and hence the design space. This is where the HASL preparation containing the potentially useful IP-integration rules is necessary to yield an efficient SoC design space exploration.

Fig. 5.4 shows an example DSE tree for three integration rules and depth of three. The root of the DSE tree is the host SoC design. The feasible changes on each SoC candidate node are obtained by application of rules, and the DSE tree is expanded to the next depth level. As can be seen all modifications on the SoC architecture are incremental, can be backtracked and can be undone by returning to the previous node.

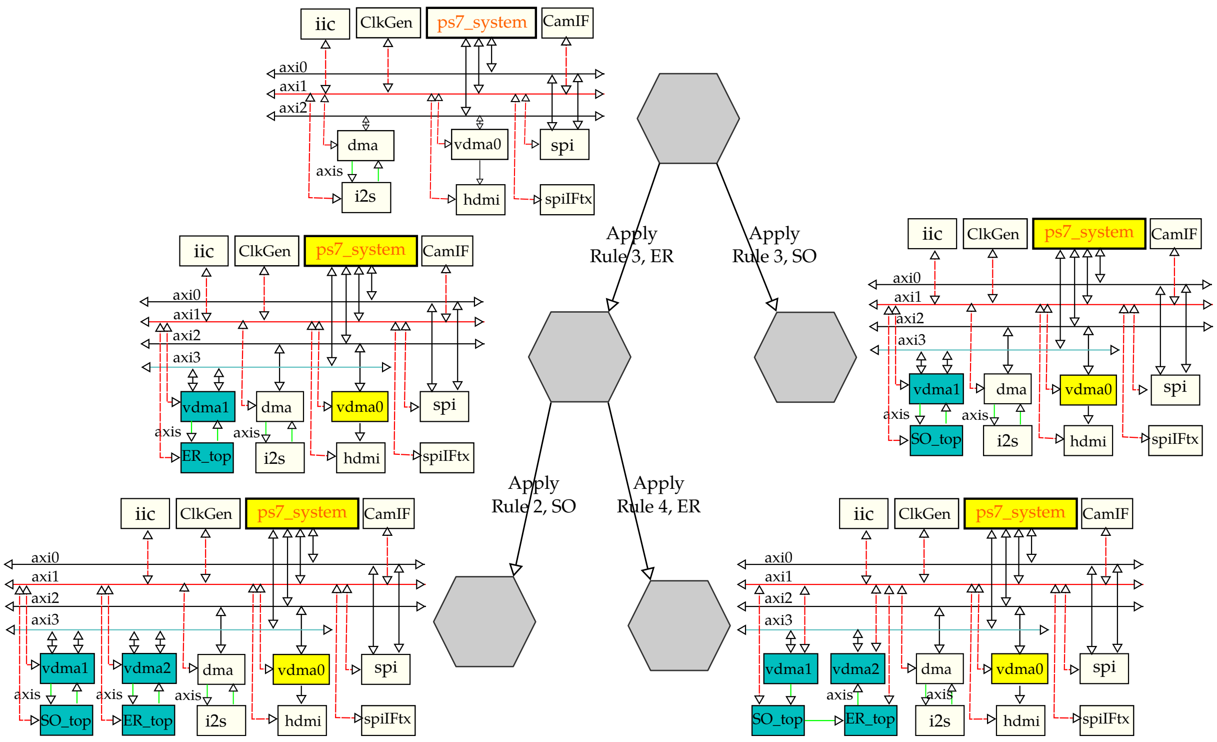


Figure 5.4.: An example design space exploration tree

As soon as this automation for the SoC bring-up and iterative structural changes is available, some of the available optimization or decision-making algorithms can be applied to achieve the desired HW-SW system design objectives. In Chapter 7, we will discuss a problem of an SoC performance monitoring on a resource-limited FPGA. In the proposed solution, the GRIP DSE engine uses an estimator of the FPGA resource utilization to provide a guidance (feedback) for selecting an appropriate IP-integration rule for progressive rule application. In the rest of this section, we will discuss tree-based SoC DSE, which mimics an SoC architect's step-by-step approach to achieve SoC optimization targeted on an IP-library.

In the proposed approach, an SoC architect can interactively search the DSE tree or select an iterative search mode. In the iterative mode, the GRIP tool considers the leaf nodes of the DSE tree as initial designs and applies all the applicable integration rules to the leaf nodes, expanding the DSE tree to the next depth level. Figure 5.5 shows the flow diagram of iterative design space exploration targeted on the HASL using the GRIP DSE engine. The DSE starts from a host SoC design, and the DSE tree is grown from the leaf SoC candidates. Firstly, a candidate SoC is picked from a list of leaf SoCs. For each selected candidate SoC, a candidate IP-integration rule is selected for the HASL and applied (if a match is found), till all the available rules are exhausted. The DSE tree is expanded width first (unless the tree is constrained, as discussed in Sec. 5.4.2). A valid rule application generates a new candidate SoC. This new SoC is appended as a new node to the DSE tree. This node is connected to the input SoC node by a directed edge (from the input SoC to the new SoC). Once all the leaf nodes of the DSE tree at a certain depth are consumed, the DSE process moves to the next set of leaf nodes at the next depth level. Fig 5.6a shows the growth of the DSE tree during the design exploration. However, many of the nodes of Figure 5.6a have identical SoC architectures, which are obtained through different sequences of rules application. Fig. 5.6b merges the nodes that have identical SoC architectures, and it shrinks the DSE tree. It can be seen that the DSE tree grows exponentially in number of nodes with the increasing width (number of rules) and the depth (successive application of rules). Here, it becomes necessary to control the exponential growth of the DSE tree, because as the tree grows, the subsequent design implementation and performance evaluation runtime for all generated candidate also grows exponentially.

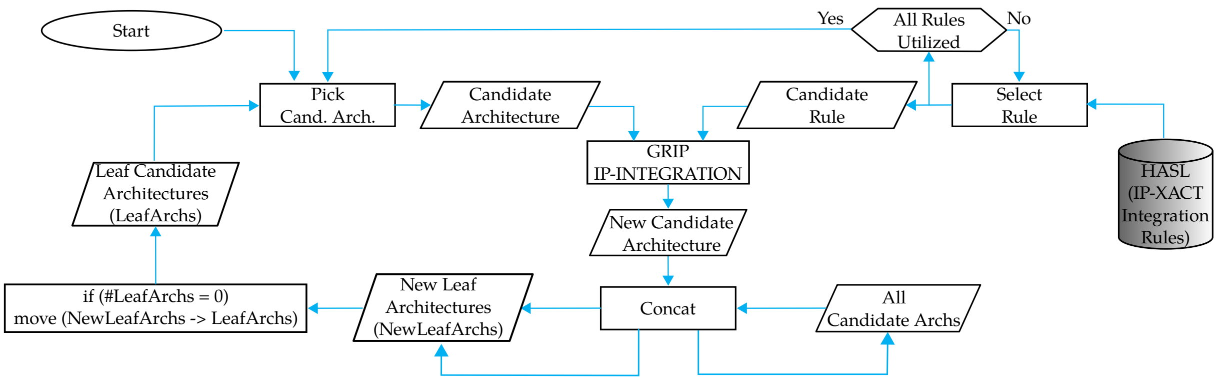

Start
Yes
All Rules Utilized
No
Pick Cand. Arch.
Candidate Architecture
Candidate Rule
Select Rule
Leaf Candidate Architectures (LeafArchs)
GRIP IP-INTEGRATION
HASL (IP-XACT Integration Rules)
New Candidate Architecture
if (#LeafArchs = 0) move (NewLeafArchs -> LeafArchs)
New Leaf Architectures (NewLeafArchs)
Concat
All Candidate Archs

Figure 5.5.: Flow diagram for SoC design space exploration using the GRIP DSE engine

5.4.2. Constraints on the Design Space Exploration Tree

In order to restrict the generation of candidate SoCs, the GRIP tool allows to interactively define constraints for the DSE tree. The constraints are defined on the search space in order to prune all non-optimal SoC candidates from the DSE tree. These constraints are defined based on the data flow of the target application, and also from the SoC architect's knowledge of the design space. The data flow of the target application gives indicators on the scope of parallelism or pipeline of execution of SW tasks. Based on this data flow, a limited set of IP-integration rules can be included in the HASL to restrict the DSE tree.

Based on the feasible SoC structural changes, we have identified a set of DSE constraints to establish a control over the DSE tree. These constraints can be used to prune the SoC candidates that fall away from the Pareto optimal front of the performance-cost plot. The GRIP tool allows the following constraints on the DSE tree:

1. Constraints on the width by limiting integration rules in the HASL.

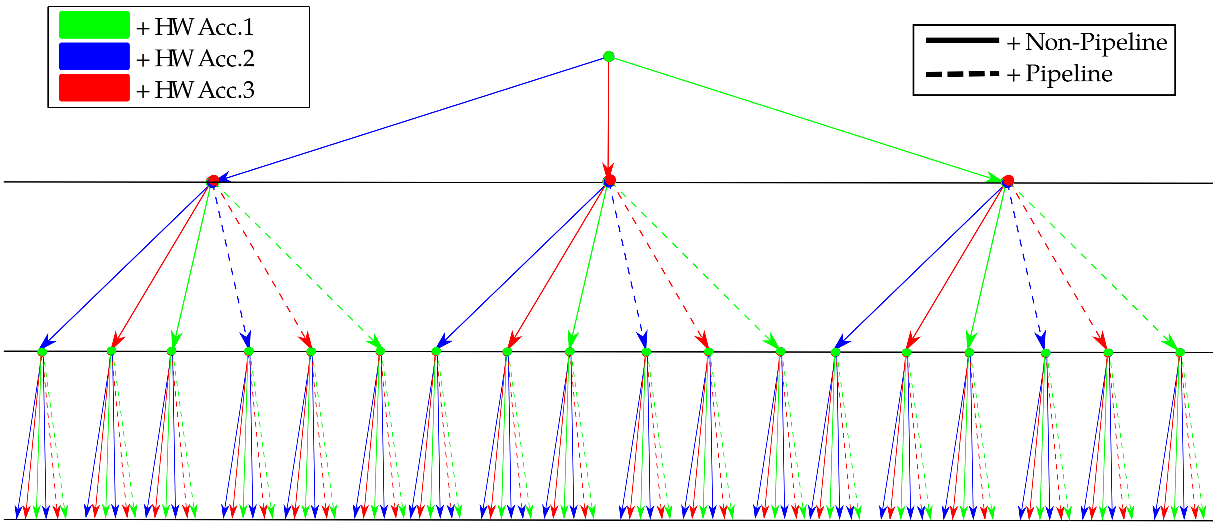
   The width of the DSE tree corresponds to the number of available IP-integration rules in the HASL, and the multiple matches of a single rule. The constraint on the width is applied by limiting the IP-integration rules in the HASL applicable for DSE, and constraint the number of matches during the rule application. Limiting the available IP-integration rules reduces the size of branching from each DSE tree node, hence the width. Fig. 5.6a and 5.6b show the DSE tree with restricted width to only three hardware accelerating subsystems.

2. Constraints on the depth by limiting the number of iterations of rules application.

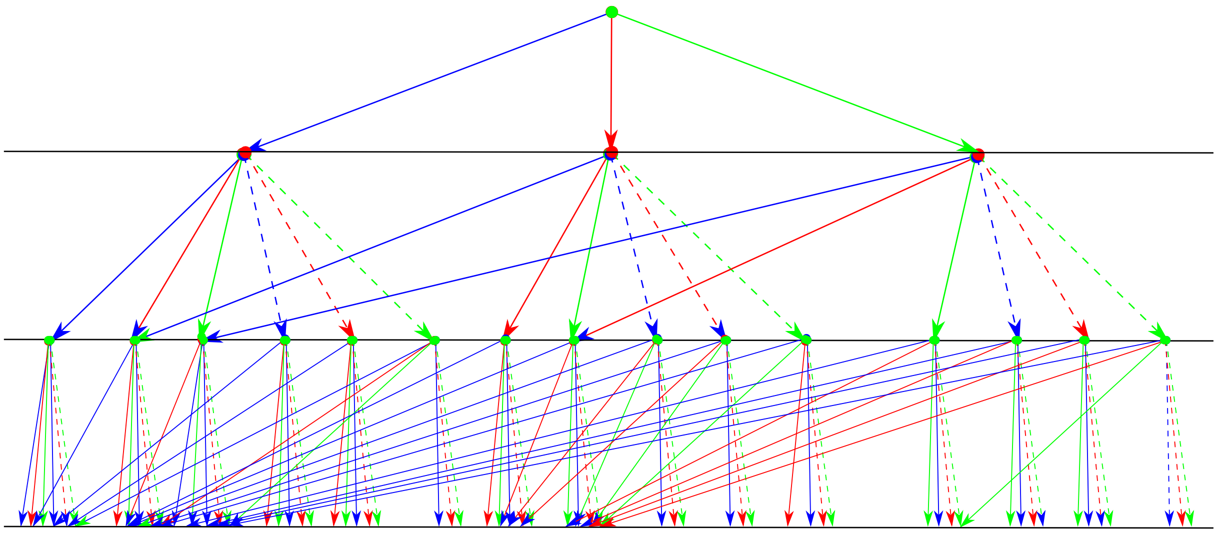
   The depth of the DSE tree corresponds to number of successive application of the rules on the host SoC. Limiting the successive rules application restricts the depth. A multiplicity of the rule application instantiates multiple IP subsystems of same kind. This improves the HW parallelism, so limiting the depth means limiting the HW parallelism. Fig. 5.6a and 5.6b show a DSE tree of depth 3.

3. Constraints on the application data flow.

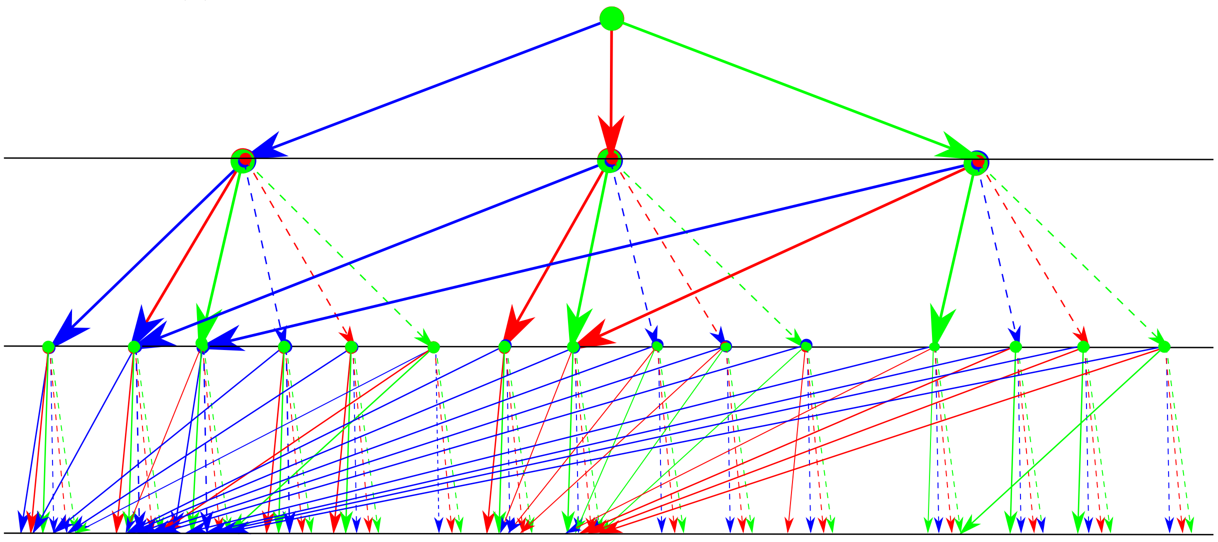
   The DSE can be restricted for either parallel or pipeline data flow. This is a derivative of the width constraint, where the HASL includes the IP-integration rules for the desired data flow. In Fig. 5.6c, the bold edges are the rule application for attaining non-pipeline SoC architectures.



(a) A DSE tree with the width of 3 hardware accelerators and the depth of 3

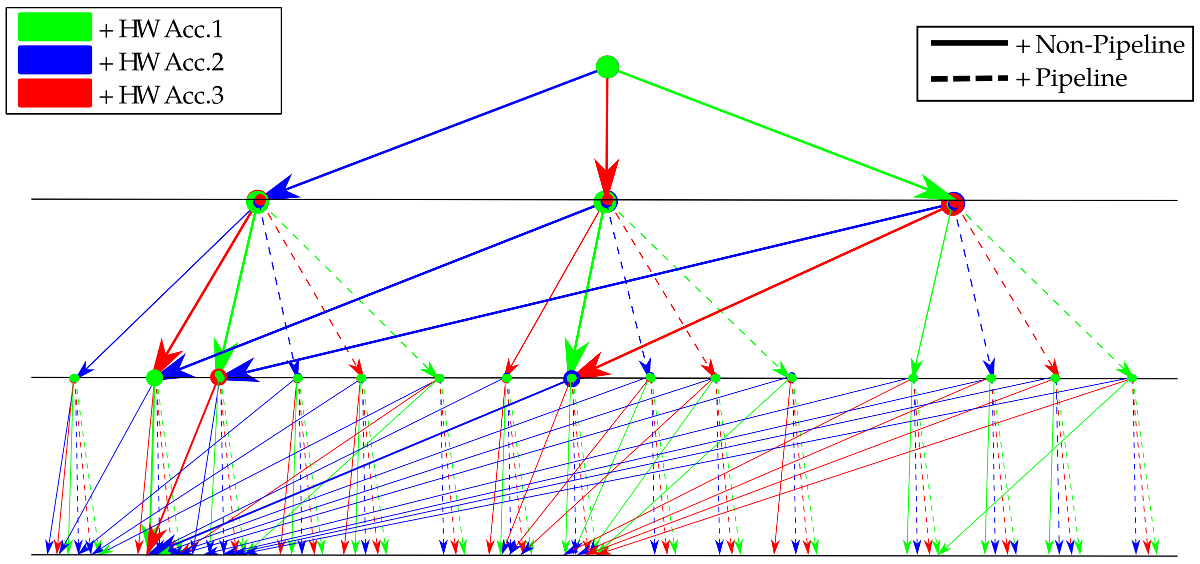


(b) A DSE tree after merging the identical SoC architecture nodes

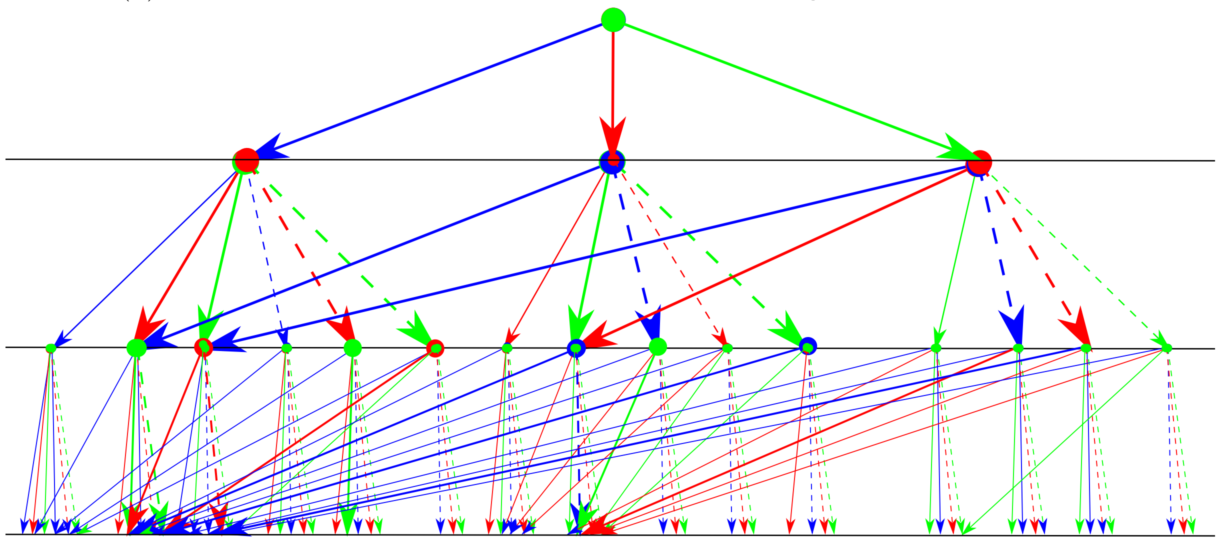


(c) A DSE tree with the task-processing constraints allowing parallel processing only

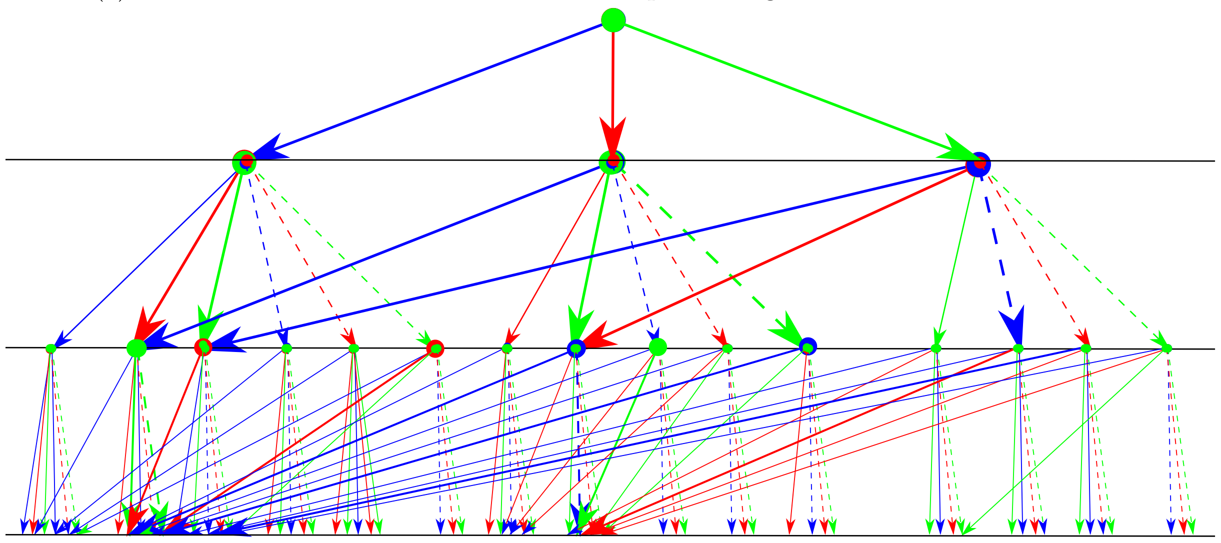
Figure 5.6.: DSE trees generation with the associated DSE constraints



(d) A DSE tree with the data-flow constraints allowing sequential data flow



(e) A DSE tree with a combination of task-processing and data flow constraints



(f) A sample DSE tree with a complex fusion of DSE constraints

Figure 5.6.: DSE trees generation with the associated DSE constraints (cont.)

4. Constraint on the task-processing schemes.

   The design space can be restricted to the sequential execution of application tasks. In a sequential application, the current processing task must finish before the next task can get started. So, the DSE must not consider multiple instantiation of a same hardware acceleration subsystem. This will still allow task pipelining, but prevent redundancy of identical hardware accelerator subsystems. The bold edges in Fig. 5.6d shows a design space for sequential data flow. By using the constraint to allow multiple instantiation of an IP subsystem enables parallelism.

Using a combination of these available constraints during DSE helps to prune the unnecessary design space. Fig. 5.6e shows an example of a design space with fusion of constraints for data flow and task processing. The bold edges show the design space for a sequential data flow, and parallel as well as pipeline data flow. Fig. 5.6f shows a design space obtained by a set of complex constraints. In this, the constraints are, a) sequential execution, b) pipeline and parallel data flow, c) sequential task-processing constraint of HW accelerator (HA) 2 follows HA1, and HA3 follows HA2.

The information for DSE constraints is generally extracted from the target software application. By analyzing the data flow of an application, the SoC architect can prepare the DSE constraints, and the HASL. Then, the GRIP engine generates all the candidate SoCs according to the specified constraints on the prepared HASL. In Chapter 6, the details on the DSE tree pruning will be discussed with two computer vision case studies.

5.5. Code Generation of HW and SW Projects for Xilinx FPGA

The target-platform code generator generates the design descriptions for a specific target platform from the IP-XACT design object of the SoC architecture. In the current state, the GRIP tool supports code-generation for Xilinx FPGAs. It generates a microprocessor hardware specification (MHS) file for Xilinx synthesis tool, this forms a part of the HW project. The generated HW drivers of the HASL form a part of the SW project. Fig. 5.7 gives an overview of the code generation steps.

5.5.1. HW Project Preparation

The target HW project contains the SoC descriptions and the HW IP library, containing all the IPs utilized in the SoC, in the target HW-platform-specific language. The HW project can be taken through the design steps of HW synthesis, performance evaluation etc. using the software tool chains supporting the target platform.

The GRIP code generator engine uses the Java file writer (Oracle 2014) to generate the MHS hardware description file. The MHS describes an SoC in term of its instantiated IP

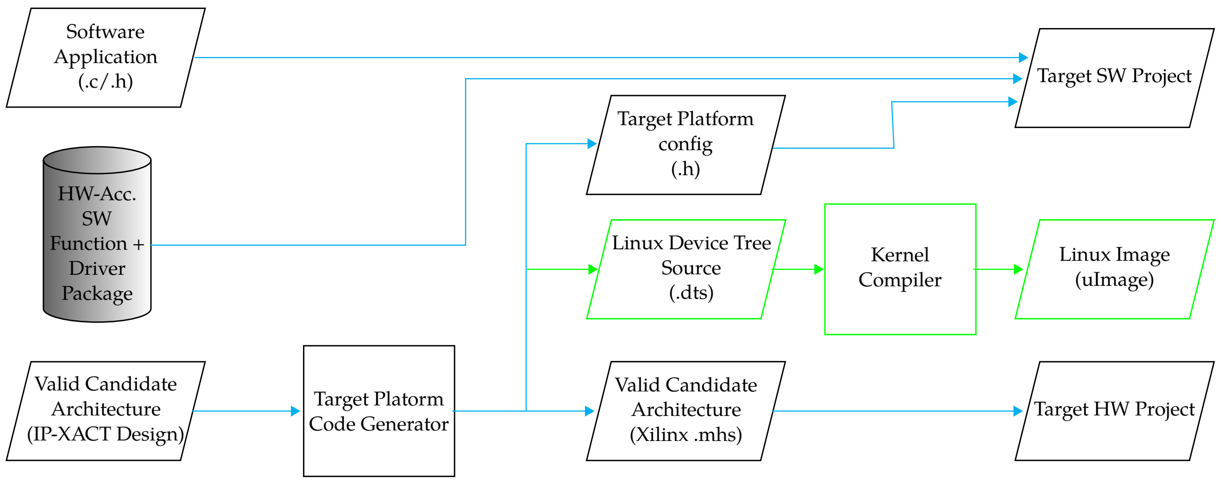

Software Application (.c/.h)
HW-Acc. SW Function + Driver Package
Valid Candidate Architecture (IP-XACT Design)
Target Platorm Code Generator
Target Platform config (.h)
Linux Device Tree Source (.dts)
Kernel Compiler
Valid Candidate Architecture (Xilinx .mhs)
Target SW Project
Linux Image (uImage)
Target HW Project

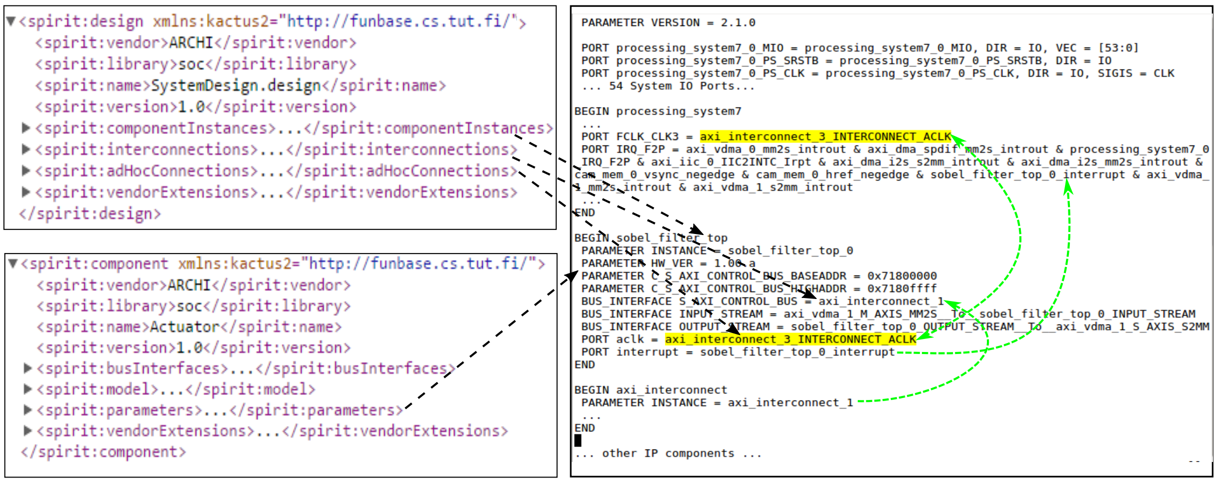
Figure 5.7.: SW and HW packages generation using the GRIP code-generation engine.

components, component parameters, and signal and bus connections among the instantiated IPs. The MHS is restricted to describe the top-level physical interfaces among IPs of an SoC, and does not consider the IP register memory maps. The mapping from the IP-XACT design description to MHS is one-to-one, except that IP-XACT describes more details on the IP register mapping which are not required in the MHS description. Fig. 5.8 describes the transformation mapping from the IP-XACT description to the Xilinx MHS description. This transformation requires both IP-XACT design and components descriptions. The code generator extracts the instantiated IPs, bus and signal interconnections from the IP-XACT design file, while it extracts the IP parameters from the IP-XACT components. The SoC top-level parameters and the IO interfaces are extracted from the IP-XACT design description.

There is one subtle difference in the IP-XACT and MHS SoC description. In IP-XACT there is no dynamic dependence among the design properties (parameters or interconnections), i.e. all the design properties can be independently handled, and any change in a certain property does not affect other properties. In the MHS description, there are few special parameters and ports (*Interrupt and power-ground*) that depend on other design properties. The code generator engine exclusively handles these scenarios as a final legalization step of the MHS generation. Fig. 5.8 shows one such dependency for the interrupt port (`IRQ_F2P`) of the CPU (`processing_system7`). It can be seen that the port `IRQ_F2P` has the list of nets separated by `"&"`. The order of the interrupt signal nets defines the interrupt priorities among the list of nets (first net being the highest priority interrupt). This list is extracted dynamically from the design interrupt signal connections. The following properties of the MHS description require post processing,

- `PORT IRQ_F2P`
- `PARAMETER C_INTERCONNECT_S_AXI_HP<*>_MASTERS`

- `PARAMETER C_USE_S_AXI_HP<*>`
- `PARAMETER C_USE_S_AXI_GP<*>`
- Power-ground nets `net_vcc and net_gnd`



```
▼<spirit:design xmlns:kactus2="http://funbase.cs.tut.fi/">
   <spirit:vendor>ARCHI</spirit:vendor>
   <spirit:library>soc</spirit:library>
   <spirit:name>SystemDesign.design</spirit:name>
   <spirit:version>1.0</spirit:version>
 ▶<spirit:componentInstances>...</spirit:componentInstances>
 ▶<spirit:interconnections>...</spirit:interconnections>
 ▶<spirit:adHocConnections>...</spirit:adHocConnections>
 ▶<spirit:vendorExtensions>...</spirit:vendorExtensions>
 </spirit:design>
```

```
▼<spirit:component xmlns:kactus2="http://funbase.cs.tut.fi/">
   <spirit:vendor>ARCHI</spirit:vendor>
   <spirit:library>soc</spirit:library>
   <spirit:name>Actuator</spirit:name>
   <spirit:version>1.0</spirit:version>
 ▶<spirit:busInterfaces>...</spirit:busInterfaces>
 ▶<spirit:model>...</spirit:model>
 ▶<spirit:parameters>...</spirit:parameters>
 ▶<spirit:vendorExtensions>...</spirit:vendorExtensions>
 </spirit:component>
```

```
 PARAMETER VERSION = 2.1.0

 PORT processing_system7_0_MIO = processing_system7_0_MIO, DIR = IO, VEC = [53:0]
 PORT processing_system7_0_PS_SRSTB = processing_system7_0_PS_SRSTB, DIR = IO
 PORT processing_system7_0_PS_CLK = processing_system7_0_PS_CLK, DIR = IO, SIGIS = CLK
 ... 54 System IO Ports...

BEGIN processing_system7
 ...
 PORT FCLK_CLK3 = axi_interconnect_3_INTERCONNECT_ACLK
 PORT IRQ_F2P = axi_vdma_0_mm2s_introut & axi_dma_spdif_mm2s_introut & processing_system7_0
 _IRQ_F2P & axi_iic_0_IIC2INTC_Irpt & axi_dma_i2s_s2mm_introut & axi_dma_i2s_mm2s_introut &
cam_mem_0_vsync_negedge & cam_mem_0_href_negedge & sobel_filter_top_0_interrupt & axi_vdma_
1_mm2s_introut & axi_vdma_1_s2mm_introut
 ...
END

BEGIN sobel_filter_top
 PARAMETER INSTANCE = sobel_filter_top_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_S_AXI_CONTROL_BUS_BASEADDR = 0x71800000
 PARAMETER C_S_AXI_CONTROL_BUS_HIGHADDR = 0x7180ffff
 BUS_INTERFACE S_AXI_CONTROL_BUS = axi_interconnect_1
 BUS_INTERFACE INPUT_STREAM = axi_vdma_1_M_AXIS_MM2S__To__sobel_filter_top_0_INPUT_STREAM
 BUS_INTERFACE OUTPUT_STREAM = sobel_filter_top_0_OUTPUT_STREAM__To__axi_vdma_1_S_AXIS_S2MM
 PORT aclk = axi_interconnect_3_INTERCONNECT_ACLK
 PORT interrupt = sobel_filter_top_0_interrupt
END

BEGIN axi_interconnect
 PARAMETER INSTANCE = axi_interconnect_1
 ...
END

... other IP components ...
```

Figure 5.8.: Mapping of IP-XACT properties to the MHS description for the HW project

After the SoC design description in the MHS format is available, it can be included in the HW project together with the available HDL description of IP components in the IP library. The design timing constraints can be configured in the HW project. Or otherwise, as in the Xilinx design flow, the design clock frequencies are included as parameters in the MHS file. This HW project can be taken through the HW synthesis in the Xilinx platform studio (XPS) tool chain. After HW synthesis, a HW bit-stream file is generated to program the FPGA. Further, the XPS generates board support package (BSP) drivers to support SW cross-compilation.

5.5.2. SW Project Preparation

The SW project consists of the custom software application and the associated HW drivers (from HASL). It also contains the board support package (BSP), which contains software code to configure the FPGA board. The BSP is generally available with the software tools support for the target board. The SW project is cross-compiled by the target platform-specific tool chain.

In the GRIP-based flow, the input SW application is not required to be modified for each SoC structural change. Instead, the underlying HASL HW drivers are adapted to the modified SoC architecture by generating the design configuration files from the IP-XACT design object. The HW drivers are generated as discussed in Chapter 4. The configuration file includes parametric information on the SoC address map, available IP subsystems (according to the applied IP-integration rules) and the register configurations for the *mode structs*. The GRIP engine uses the design configuration files and the hardware drivers from the IP library to complete the target SW project. The SW project is built in the Xilinx Software Development Kit (SDK) tool chain to generate the SW binary (ELF) for the FPGA. Figure 5.9 shows the design flow in the Xilinx tool chain from the HW and SW projects to generate the binary files (BIT for HW and ELF for SW) for the FPGA.

If a SW application has to run on Linux OS, the code-generator also generates the *device tree source (dts)* files for the kernel. The integration of the GRIP tool to generate the Linux kernel will be discussed in Sec. 5.7.

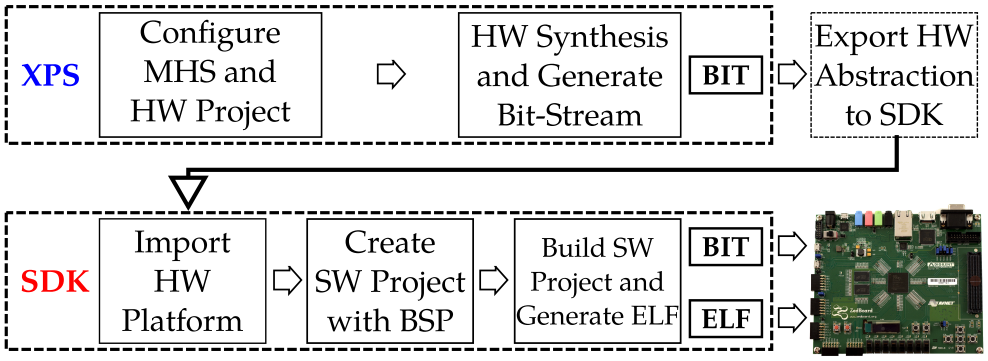

XPS
Configure MHS and HW Project
HW Synthesis and Generate Bit-Stream
BIT
Export HW Abstraction to SDK
SDK
Import HW Platform
Create SW Project with BSP
Build SW Project and Generate ELF
BIT
ELF

Figure 5.9.: Xilinx XPS and SDK tools to generate respectively the HW and SW binaries

5.6. GRIP Tool Integration with the Xilinx Toolchain

Fig. 5.10 illustrates a possible integration of the GRIP tool into the Xilinx design flow. The GRIP tool is called before the standard design tools and supplies SoCs candidates with HW acceleration according to the SW definition of the SoC given by the input application. These candidates are forwarded to the code generation engine to obtain the HW and SW project for the target platform (in our case, Xilinx Zynq FPGA).

In this work, the GRIP tool is used together with the Xilinx ISE tool chain, including *Xilinx platform studio (XPS)* for the HW project and *Xilinx software development kit (SDK)* for the SW project. The GRIP code generator transforms the IP-XACT SoC descriptions to the corresponding MHS descriptions. The generated candidates are taken through the synthesis steps using the XPS. The synthesis process generates the HW binary file for an FPGA, also called bit-stream file. The SDK cross-compiles the SW applications for the ZedBoard FPGA (bare metal version). The SDK takes the `C/C++` implementation of a software application and the HW drivers, together with the board support package (BSP), and cross-compiles it for the targeted ARM core executable binary code (ELF file). These SW and HW binaries are used to program the FPGA for system prototyping.

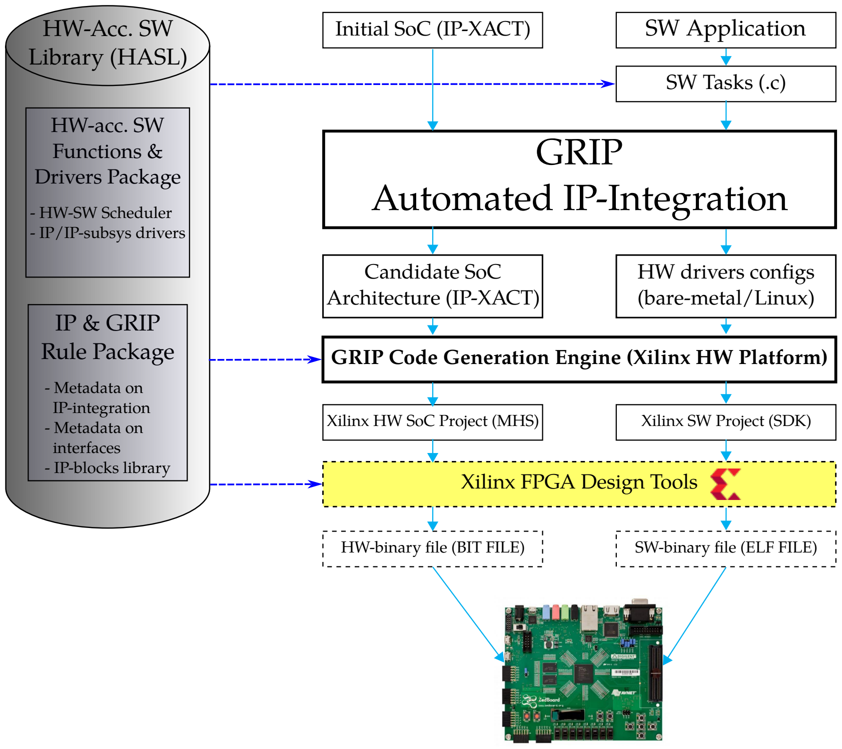

HW-Acc. SW Library (HASL)
HW-acc. SW Functions & Drivers Package
- HW-SW Scheduler
- IP/IP-subsys drivers
IP & GRIP Rule Package
- Metadata on IP-integration
- Metadata on interfaces
- IP-blocks library
Initial SoC (IP-XACT)
SW Application
SW Tasks (.c)
GRIP
Automated IP-Integration
Candidate SoC Architecture (IP-XACT)
HW drivers configs (bare-metal/Linux)
GRIP Code Generation Engine (Xilinx HW Platform)
Xilinx HW SoC Project (MHS)
Xilinx SW Project (SDK)
Xilinx FPGA Design Tools
HW-binary file (BIT FILE)
SW-binary file (ELF FILE)

Figure 5.10.: Xilinx FPGA SoC prototyping using the HASL in the GRIP environment

5.7. GRIP Tool Integration with the Linux OS

In the case of Linux OS based SoC prototyping, in addition to the HW and SW projects generation, it also requires the Linux build process to be integrated with the GRIP tool. Fig. 5.11 describes the integration of GRIP generated descriptions with the Xilinx tools and Linux build process. The Linux OS has four main components in the kernel space:

a) Boot.bin, b) Device tree blob (DTB), c) uImage, d) Linux File system.

Boot.bin contains the HW bit-stream file, `uBoot`, and the first stage boot loader (FSBL). The HW project is prepared and synthesized in the Xilinx XPS tool to generate the HW bit-stream file. The `uBoot` (Universal Boot Loader) contains the set of instructions to boot the OS kernel. It initializes the memory table, drivers required for different peripherals, and starting-up of the Linux kernel. It is built using the scripts provided as a package for Linux kernel compilation (Xilinx 2012) (generally using `Makefile` scripts). The FSBL provides the target platform initialization routines. The FSBL is built using the Xilinx SDK, after the HW system is synthesized and corresponding HW abstraction is available from the XPS project. Finally, all three - `FSBL, BIT file, uBoot` - are bundled together in the SDK to create the `Boot.bin` file.

The DTB contains the system address map and *compatibility string* based mapping of the IP components to the corresponding kernel space drivers. It is built directly from the device tree source (DTS) files using the cross-compile scripts available in the Linux kernel package.

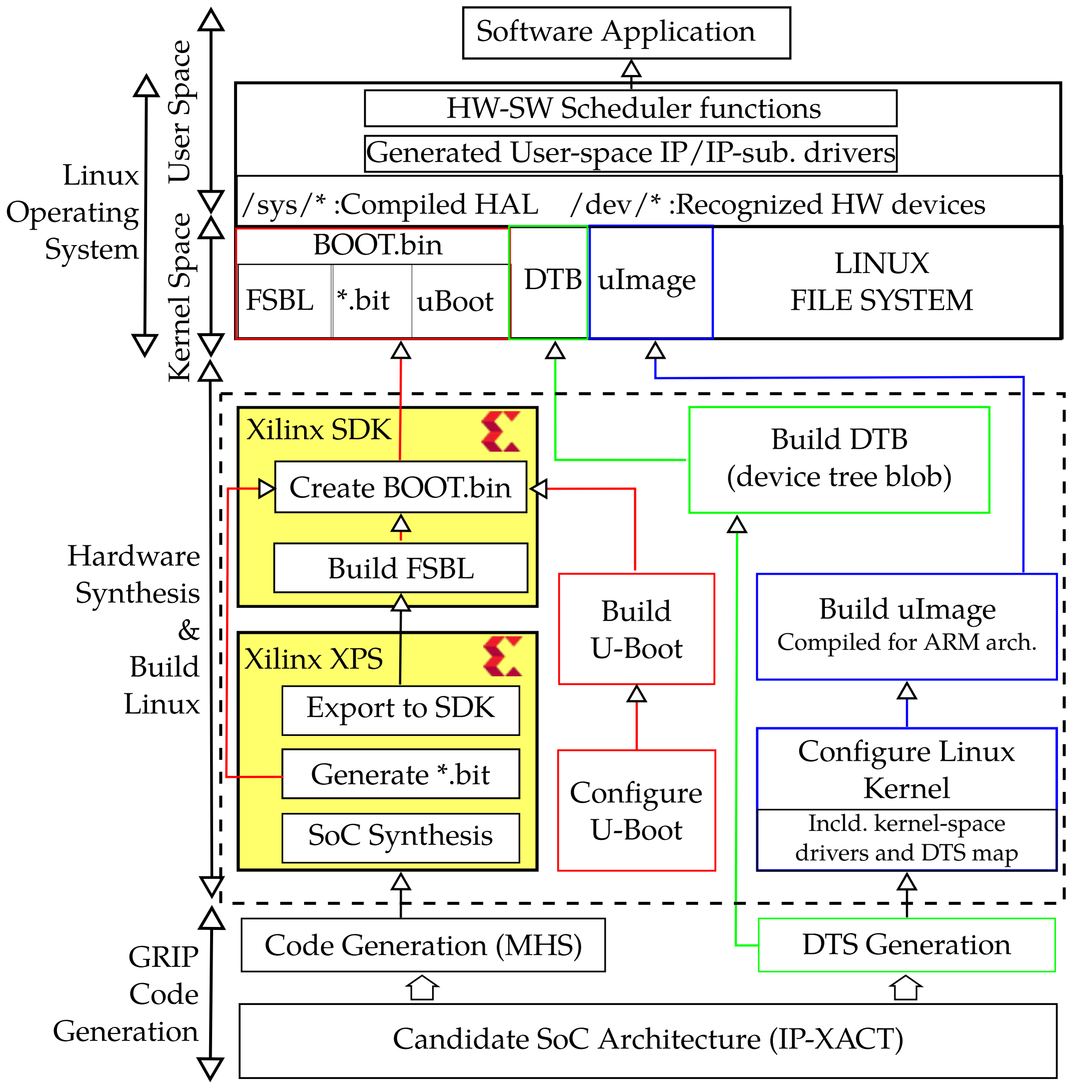


Figure 5.11.: Building the Linux kernel using GRIP generated HW drivers

The uImage is the compressed Linux kernel image for a specific CPU architecture. It includes the HW drivers corresponding to the kernel configurations specified during the build process. It is generated from the Linux kernel build routines. This work only uses memory-mapped IP subsystems for hardware acceleration, the *character* (`char`) device drivers were included as kernel-space drivers while building the uImage.

The Linux file system is the file system image of the Linux operating system. The Linux operating system has a well defined structure for the file system. The file system contains user files, application programs, administrative files, shared libraries etc.. In this work, we use version 12.04 of the `ubuntu` file system. The file system is not affected by any structural changes at the HW architecture level.

In the kernel space, for the integration of the GRIP tool with the Linux build, the GRIP tool generates the the MHS and DTS files from IP-XACT SoC descriptions. The MHS file goes through HW synthesis in the Xilinx tool chain to generate the FSBL, bit file, and uBoot as the `BOOT.bin`. The DTS file is built to create the `DTB` file. While the configurations in the `DTB` file are used to link the kernel space drivers to the HW abstraction in DTB, and build the `uImage` Linux kernel.

Finally, in the user space, the GRIP generated *HW drivers*, together with the *Linux utility functions* form the `user-space IP-subsystem drivers`. These user space drivers use the predefined system calls to access the kernel space (Code 4.10, Code 4.11) These drivers provide the functions for the software applications to access the kernel space, and the HW resources.

5.8. Conclusions

In this chapter, we discussed the details of the three core engines of this work: the IP-integration engine, the DSE engine, the code generation engine.

The IP-integration engine uses the IP-integration rules of the HASL to automate SoC structural changes. This is extended by the GRIP DSE engine, which iteratively applies the IP-integration rules to explore the SoC design space. This mimics the SoC architect's step-by-step DSE approach, and the structural changes can be backtracked and undone. The SoC architect can control the automated DSE process by applying the DSE constraints. Finally, the code generation engine generates HW and SW projects for the targeted HW platform. We also discussed the integration of the GRIP tool with the external platform-specific software tool chains and with the Linux OS build process.

The chapter extends the available graph-grammar principles to the model-based graph modeling and graph rewriting. The algorithms used for the IP integration and DSE are also adapted to the model-based frameworks, while they use OCL, EVL, and EOL frameworks.

6. Computer-Vision Case Studies on the ZedBoard

"...Mapping a design to FPGA prototype hardware can also be time-consuming and prone to error. When a design does not work in a prototype, it can be because of physical problems, design errors, or mapping issues. Without good techniques and the necessary tools, bringing up a prototype board can add months to your project schedule.", Ron Green, Technical Communications Manager, S2C Inc.

6.1. Introduction

In the previous chapters, we have described the key elements of the GRIP tool, and proposed an SoC design methodology using the GRIP tool. This chapter demonstrates the proposed design approach using GRIP on two computer vision (CV) case studies: a motion detection application and a video filtering application. We will take these computationally intensive applications and explore the architectural design space to accelerate the applications. The DSE uses domain-specific HASLs containing hardware accelerators for computer vision functions. The GRIP tool is utilized to prepare the HASL and corresponding IP-integration rules. This chapter will discuss the step-by-step approach of performing the proposed GRIP tool based DSE.

In the presented case studies, the HW and SW projects are targeted to be programmed on the Digilent ZedBoard$^{\text{TM}}$ FPGA development board. The bring-up of an FPGA is tedious, it requires the implementation of a synthesizable HW design and a compilable SW application with valid HW drivers. Typically, the implementation using the available "drag-and-done" based tools require multiple iterations of the HW and SW projects for FPGA bring-up. Depending on the complexity of the HW design and the SW application, an FPGA bring-up can take up to few weeks of effort. The GRIP tool off-loads the manual effort to instantiate, configure and integrate IP components into an SoC for the HW project, and the implementation of HW drivers for the SW project. The proposed GRIP-based IP-integration and DSE methodology requires one-time effort to prepare IP-integration rules associated to each IP component. However, the atomic structural changes encoded by each IP-integration rule can be iteratively utilized to explore a much bigger design space than practically feasible using manual schemes.

6.2. Host SoC on the Zynq FPGA

The design space exploration for both CV applications starts from the host SoC architecture. This host SoC contains the minimal HW setup to execute a general computer vision application in the pure software implementation without any hardware acceleration. The DSE is targeted on the ZedBoard FPGA evaluation board with Zynq FPGA SoC chipset (ZedBoard 2012).

The host SoC design contains IP components for IO interfaces in the processing system (PS), and the camera and HDMI interface IPs in the programmable logic (PL) part of the Zynq chipset, specifically:

1. An ARM cortex_a9 Dual-CPU (operating at 667MHz, *ps7_system*).
2. A 512 DDR2 RAM (operating at 566MHz).
3. (with Bare metal) An external camera interface via PMOD ports of the ZedBoard, configured by an I2C IP (10fps, VGA 640x480, RGB556, 100MHz System Clock).
4. (with Bare metal) An HDMI interface module for connecting an external monitor to observe the output (using VDMA at 200MHz).
5. (with Linux OS) A USB IP interface in the PS for connecting an external USB camera (10fps, VGA 320x240).
6. (with Linux OS) An HDMI interface module in the PS for connecting an external HDMI Monitor.

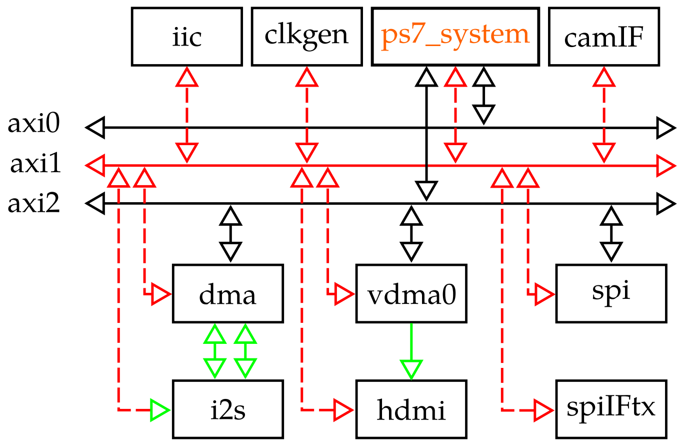


Figure 6.1.: The host SoC with associated IP components and bus interconnections

Fig. 6.1 describes the block diagram of the host SoC with the minimal IP components required for software only implementation of a computer vision application. It contains a CPU (*ps7_sys*), camera interface (*camIF*), HDMI interface (*hdmi*), clock generator (*clkgen*), and other interface modules for configuring the peripherals. Fig. 6.2 describes the block diagram of the host SoC in the context of the Zynq chipset. The figure shows the instantiated IP components and their corresponding bus interconnections. In the PL, `axi_1` is the AXI-LITE bus, `axi_0` and `axi_2` are the AXI4 buses, while AXI-Stream buses are highlighted with green color.

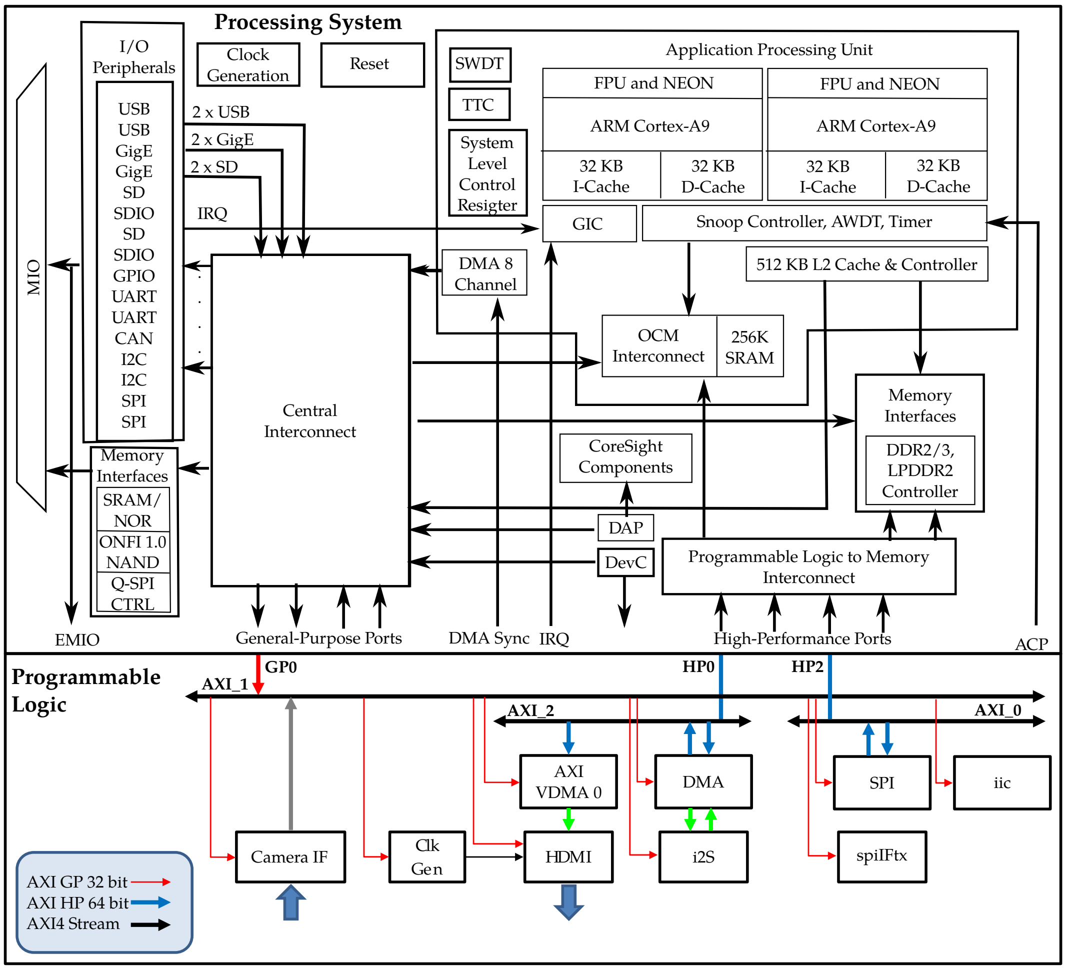


Figure 6.2.: Implementation of the host SoC in the Zynq chipset

6.3. Case Study 1 - Motion Detection Application

The motion detection application considered in this case study is based on Optical Flow (Stein 2004). In this application, the motion of the objects in the captured video is detected and shown as motion vectors on the display monitor. It uses census transformation algorithms for detecting relative motion of objects in alternate frames. The use cases of the motion detection algorithms can be seen for motion segmentation, 2D or 3D object shape recognition, alignment and calibration, video compression etc.

Fig. 6.3 describes data flow of the motion detection application. The application consists of four main SW tasks,

1. Census transformation
2. Matching signatures
3. Ego-motion estimation
4. Ego-motion compensation

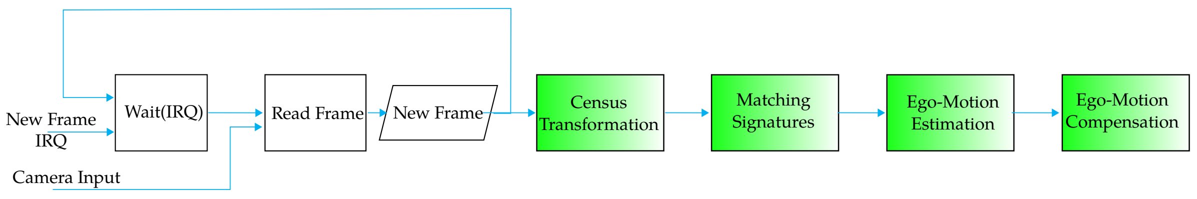


Figure 6.3.: Data flow of the motion detection application

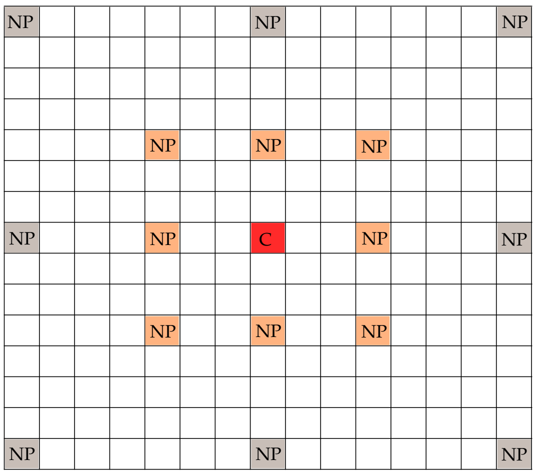
Census transformation

Census transformation transforms each pixel of an image frame in proportion to its intensity value in the range 0 to 255. In the simplest form, each pixel is thresholded to obtain a binary 0 or 1 value. In a slightly more complex implementation, each pixel can be thresholded in the range of intensities to obtain 2-bit encoding, 00, 01, 10, or 11.

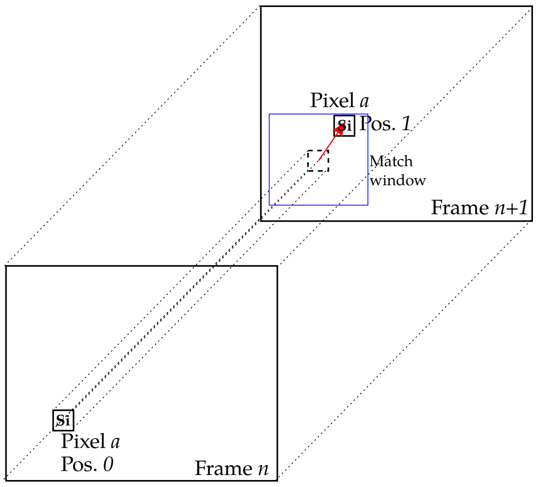
As a first step, census transformation is used to obtain a signature value for each pixel of an input frame. These signatures for individual pixels are then used in the next step (matching) to find the corresponding pixels across alternate frames. For a candidate pixel, a signature is formed by the concatenated census transformed values of neighboring pixels. In the simplest form, the candidate pixel is considered together with a 3x3 pixels window with the candidate pixel at the center (3x3 - 1 = 8 bit signature). In this case study, a more complex signature of a 15x15 window is used around a candidate pixel to generate a 32 bit signature (Fig. 6.4a). The signature is formed by concatenating the 16 neighboring pixels (NPs) as shown in Fig. 6.4a, with each NP census transformed to a 2-bit value. The signature width defines the trade-off of accuracy of the matching step and the computation time.

Matching signatures

The signature matching is used to match pixels with identical signatures across alternate frames. It is a window based operation, where a candidate pixel signature is matched to all the pixels in a *match window* (Fig. 6.4b). For each pixel that is found moved across alternate frames, a motion vector is generated representing the motion of the pixel. The motion vector indicates the effective displacement of a candidate pixel. This is computationally the most expensive step of the motion detection application. The processing load depends exponentially on the *match window*.



(a) Signature generation using census transformation (15x15 window, C = Candidate pixel, NP = Neighbouring pixel). "Thresholding" is done specifically for the 16 NPs



(b) Signature matching across alternate frames restrained to a match window

Figure 6.4.: Signature generation and matching signature steps for motion detection

Ego-motion estimation

This step together with the next step is required to tackle the scenario of video captured by a moving camera device. In this step the effective motions of the camera device is estimated using the *First Order Flow (FOF)* model (Paul 2010) (Lyu 2016). The FOF model considers three motions, namely, a) Dilation element (D), b) Rotation element (R), c) Shear element (S). The dilation effect is because of moving in and moving out of camera w.r.t. an object. The rotation element comes from the rotation of the camera along the vertical axis. The shear effect is because of lateral movement of the camera.

Equation 6.1 describes the FOF model applied on a pixel at (X, Y) coordinates. In the equation V_x and V_y are the X- and Y-components of the optical flow motion vector V on the pixel at (X, Y) coordinates. In Eq. 6.1, θ represents the rotation along the vertical axis, and X_c and Y_c are for the focus of expansion of dilation.

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} D+S & -R \\ R & D-S \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X-X_c \\ Y-Y_c \end{bmatrix} \tag{6.1}$$

A simplification is applied to this equation, θ is set to zero degrees assuming only lateral motion of the camera, and shear element (S) is set to zero with an assumption of slow moving camera device. Eg. 6.2 gives the simplified FOF model used in computation.

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} D & -R \\ R & D \end{bmatrix} \begin{bmatrix} X - X_c \\ Y - Y_c \end{bmatrix} \tag{6.2}$$

Eq. 6.2 has four unknowns (D, R, X_c, Y_c) and the matrix gives two equations. In order to find the model unknowns, two randomly selected pixels are used in Eq. 6.2 to form four equations (assuming that the motion of the camera device has same motions effect on most of the frame pixels). If (x_1, y_1) and (x_2, y_2) are the two pixels with corresponding motion vectors (v_{x1}, v_{y1}) and (v_{x2}, v_{y2}), then the FOF model unknowns can be found using the equations 6.3. Further, Ego-estimation engine (EEE) tries multiple iterations to find the best FOF model. The best FOF model is selected based on the *outlier rate*, which is the ratio of motion vectors that do not fit the model and total number of motion vectors. The EEE fails to generate the FOF model, if the *outlier rate* is more than a minimum required outlier rate.

$$\begin{aligned}
X_c &= \frac{e_1 + e_2 + e_3}{v_{x1} \times (v_{x1} - 2 \times v_{x2}) + v_{x2}^2 + v_{y1}^2 - v_{y2} \times (2 \times v_{y1} - v_{y2})} \\
Y_c &= \frac{e_4 + e_5 + e_6}{v_{x1} \times (v_{x1} - 2 \times v_{x2}) + v_{x2}^2 + v_{y1}^2 - v_{y2} \times (2 \times v_{y1} - v_{y2})} \\
D &= \frac{(x_1 - x_2) \times (v_{x1} - v_{x2}) + (y_1 - y_2) \times (v_{y1} - v_{y2})}{x_1 \times (x_1 - 2 \times x_2) + x_2^2 + y_1 \times (y_1 - 2 \times y_2) + y_2^2} \\
R &= \frac{(x_1 - x_2) \times (v_{y1} - v_{y2}) + (y_2 - y_1) \times (v_{x1} - v_{x2})}{x_1 \times (x_1 - 2 \times x_2) + x_2^2 + y_1 \times (y_1 - 2 \times y_2) + y_2^2}
\end{aligned}$$

Where e_1 to e_6 are:

$$\begin{aligned}
e_1 &= v_{x1} \times (-v_{x2} \times x_1 + v_{x1} \times x_2 - v_{x2} \times x_2 + v_{y2} \times y_1 - v_{y2} \times y_2) \\
e_2 &= v_{y1} \times (-v_{y2} \times x_1 + v_{y1} \times x_2 - v_{y2} \times x_2 - v_{x2} \times y_1 + v_{x2} \times y_2) \\
e_3 &= x_1 \times (v_{y2}^2 + v_{x2}^2) \\
e_4 &= v_{x2} \times (v_{y1} \times x_1 - v_{y1} \times x_2 - v_{x1} \times y_1 + v_{x2} \times y_1 - v_{x1} \times y_2) \\
e_5 &= v_{y2} \times (-v_{x1} \times x_1 + v_{x1} \times x_2 - v_{y1} \times y_1 + v_{y2} \times y_1 - v_{y1} \times y_2) \\
e_6 &= y_2 \times (v_{x1}^2 + v_{y1}^2)
\end{aligned} \tag{6.3}$$

Ego-motion compensation

At this step, the FOF model parameters as computed in the previous step are used to compute the effective optical flow accounted to the moving camera device. The motion vectors generated during the matching signature step are subtracted by the motion vectors obtained from the best Ego-estimation FOF model. Out of the four SW tasks, this is computationally the least expensive SW task.

A SW library is prepared which contains the `C/C++` implementation of these functions for the motion detection application. The library is available both for the bare metal as well as Linux OS based motion detection application development. The implemented functions in the library are targeted for the ARMv7 architecture of the ARM cortex-A9 CPU (cross-compilable using the ARM GCC compiler).

In the next section, we will discuss using this SW library, together with the HW implementation of previously described four functions of the motion detection application to prepare the HASL. The prepared HASL is then employed for the GRIP design space exploration.

6.4. GRIP DSE for the Motion Detection Application

The objective is to implement the software only version of the motion detection application on the host SoC architecture, and progressively improve the application processing time by iteratively transferring the SW tasks from the CPU to the hardware accelerators. The effective change in the application processing is the cumulative effect of SW tasks performance improvement by HW acceleration and associated data communication overhead. In the subsequent sub-sections, we will discuss the steps to perform design space exploration using the GRIP tool.

6.4.1. HASL Preparation for the Motion Detection Application

The first step in the proposed approach is to prepare a domain-specific HASL for the target application. As discussed in the previous section, the motion detection application has four main computationally expensive SW tasks. So, these SW tasks offer the scope of HW acceleration in the application. This requires the availability of hardware accelerators in the IP library for each of the SW tasks.

IP Packaging: The four hardware accelerators corresponding to the SW tasks are implemented by using the Xilinx high level synthesis (HLS) tool, Vivado-HLS (Xilinx 2014). The synthesizable software implementation (`C/C++`) of each SW task is given as input to the Vivado-HLS. These SW descriptions include HLS directives, which enable HW optimizations within the Vivado HLS tool, including *pipelining, loop unrolling, HW parallelization* etc. Inclusion of these directives require analysis of data flows within each SW task, while making a trade-off among HW resources and performances. The HLS is used to quickly implement the hardware accelerators. However, the GRIP approach doesn't rely on HLS. The HW IPs available for this DSE include:

1. Hardware accelerator (HA) for the census transformation (CE)
2. HA for the matching signature (ME)
3. HA for the Ego-motion estimation (EEE)
4. HA for the Ego-motion compensation (ECE)
5. Other IPs for camera & display interfaces

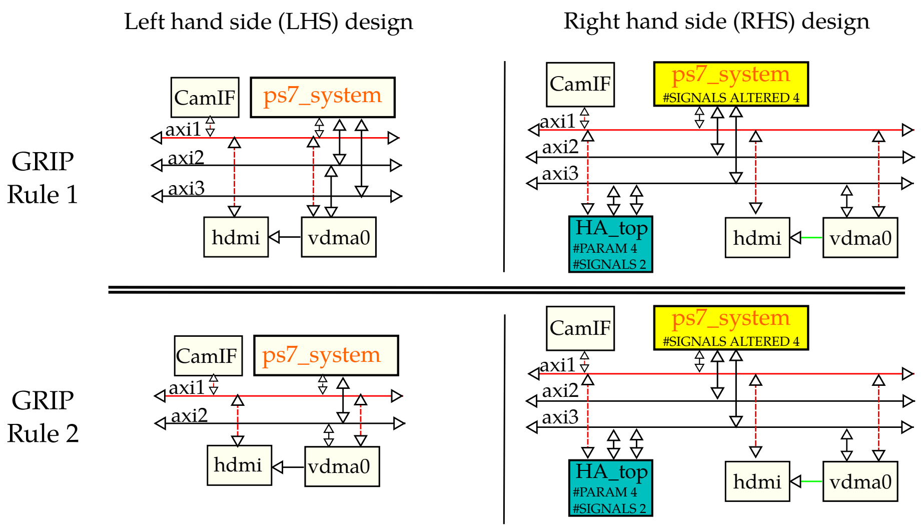
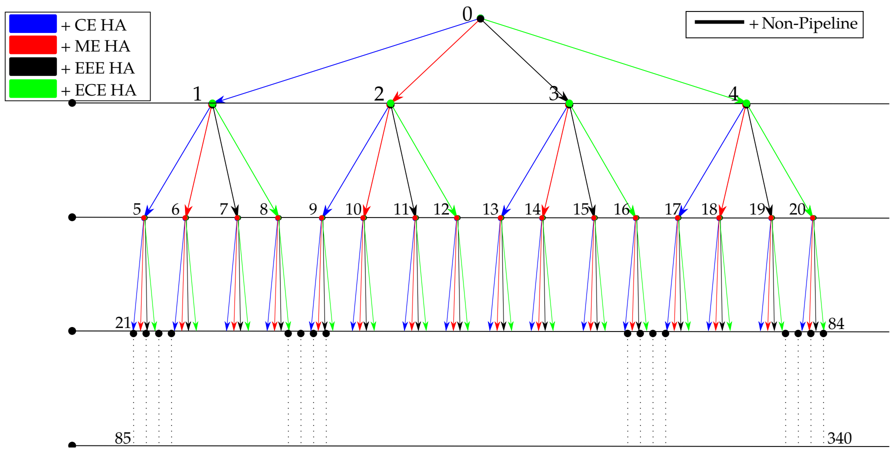


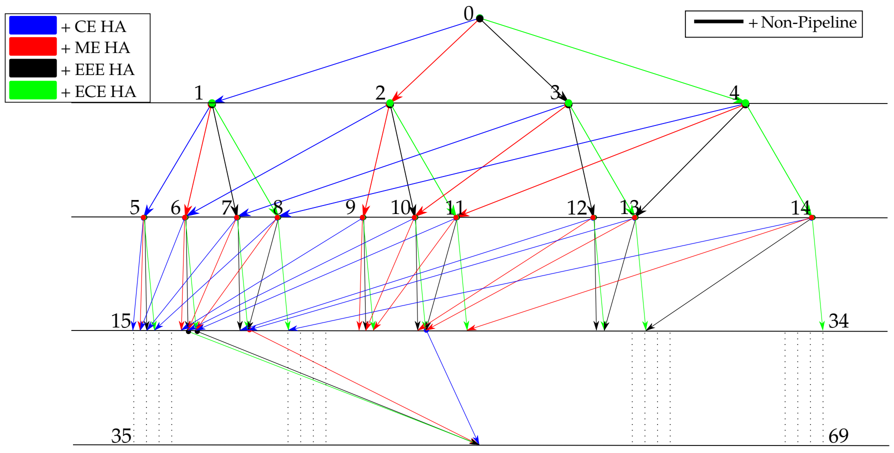
Figure 6.5.: IP-integration rules for the motion detection application

For each accelerator IP (CE, ME, EEE, ECE), the integration rules are prepared for integration on a dedicated new bus, and on an existing bus (Fig. 6.5). Thus, in total eight IP-integration rules for all HA IPs are prepared. The IP-XACT components together with their HDL descriptions and the associated IP-integration rules make up the IP packages for our IP library.

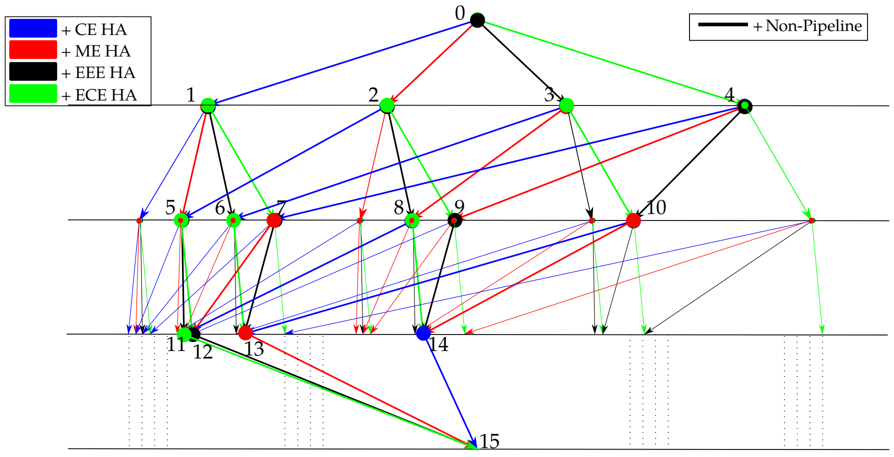
HW-accelerated SW library (HASL): Using the HASL drivers generator, the required HW drivers and schedulers are generated. The available IP-XACT hardware meta-model information is utilized to generate the driver and scheduling layer. For each of the four motion detection application tasks, the HASL also has a SW implementation. Thus, all four SW tasks can be executed either on the HW accelerator subsystem or on the ARM CPU.



(a) DSE tree with no DSE constraints



(b) DSE tree with merged nodes and no DSE constraints



(c) DSE tree with merged nodes and with DSE constraints

Figure 6.6.: DSE trees for the motion detection with and without the associated DSE constraints (width of 4 hardware accelerators and tree depth of 4)

6.4.2. DSE for the Motion Detection Application

The DSE is an interactive process to explore the performance and utilized HW resources trade-off. First, we run a broad search with a depth of four (Fig. 6.6a). Node 0 is the initial SoC with accelerator IP, node 1 to 4 have one IP for CE, ME, EEE, and ECE. Progressively more IPs are added to reach higher tree depth. It is clear from the figure that the number of DSE tree nodes grows exponentially. However, the SoC candidate architectures represented by the DSE tree nodes are not unique. At each DSE tree depth, the nodes with identical SoC architectures can be merged into a single node. In the DSE tree with depth four, after merging similar nodes, there are 69 feasible candidate SoCs (Fig. 6.6b). Fig. 6.7 shows the Zynq-chipset block-diagram for the SoC candidate architecture with all hardware accelerators instantiated, CE, ME, EEE, and ECE. The figure also shows the task mapping on the HW processing units. The application control and task mappings are handled by the generated HW drivers from the HASL

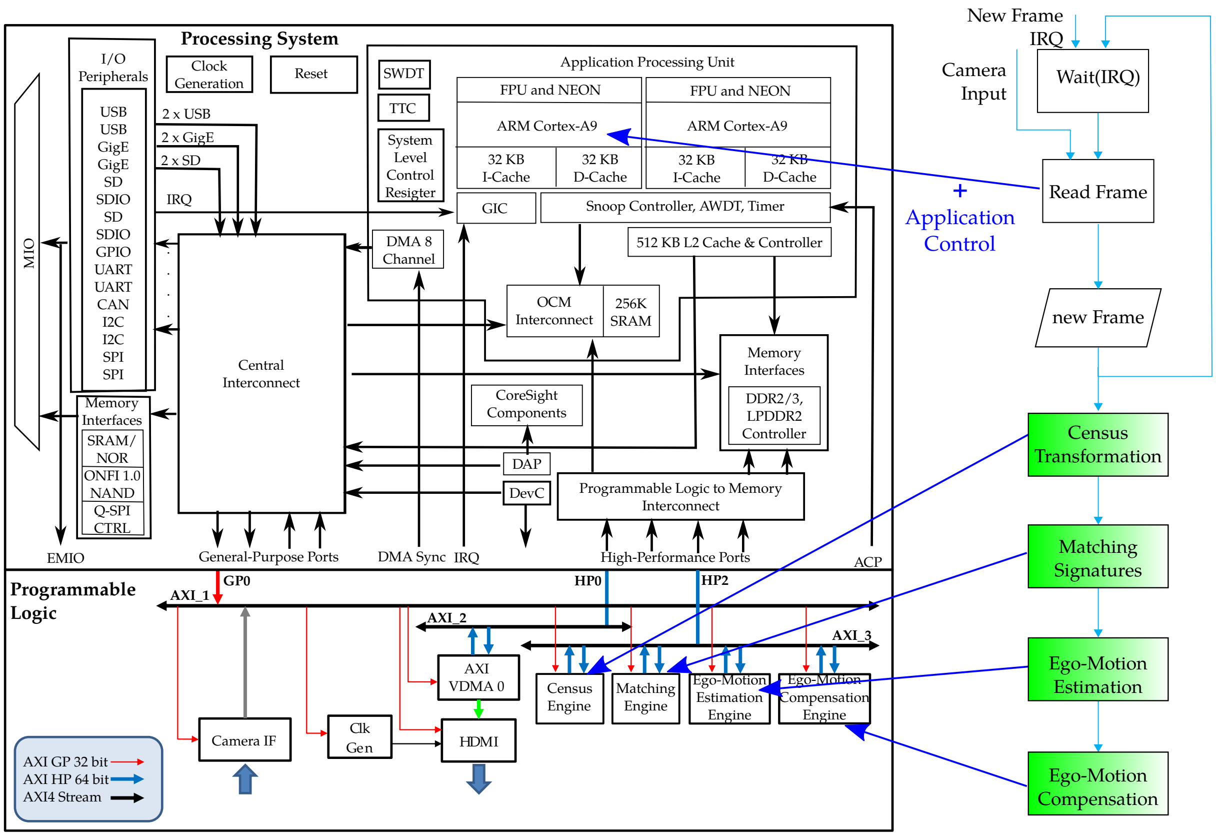


Figure 6.7.: The candidate SoC with all four HAs for motion detection on Zynq-chipset

DSE Constraints: The FPGA synthesis for all 69 candidate SoCs would take very long if done sequentially, or would require many computing resources if done in parallel. So, we apply additional constraints on the DSE tree synthesis in order to prune non-optimal candidate SoC architectures. We look at the data flow of the application (Fig. 6.3) to extract the following DSE constraint: Restrict to single instantiation of each accelerator IP: Application has sequential SW tasks execution on each frame. Architectures with parallel computation on two IPs of same type bring no benefit, as at the max one IP is active at any given time.

After applying this DSE constraints, the number of candidate SoCs is reduced to 15, which are shown by the bold edges in Fig. 6.6c. DSE constraints require understanding on the application, otherwise they may prune pareto-optimal candidates.

Code Generation: The GRIP code-generation engine generates the platform-specific hardware description files, which are then taken through Xilinx SoC synthesis with available RTL IP components, to generate a *bit-stream* file for the FPGA hardware programming. For the SW project, the application uses the generated HW-SW scheduler & IP-subsystem drivers and cross-compiled to generate the *SW binary* file, and is loaded to the instruction memory of the CPU. Fig. 5.11 describes the steps for building Linux-OS and including user-space drivers. The code-generation engine generates the device-tree source (DTS) for building device-tree blob (DTB). The HW devices are recognized by Linux from the DTB and appear under */dev/**. The application can access these drivers using generated user-space drivers and utility functions.

6.4.3. Results Analysis

In the previous subsection, all the generated MHS descriptions are taken through Xilinx synthesis runs to generate the bit-stream files. Each of the synthesis runs takes around 60 mins to complete. However, all the synthesis runs can be executed in parallel. Table 6.1 summarizes the architectures of all 15 generated candidate SoCs. The table describes for each candidate SoC which SW task is executed on a CPU or by a corresponding hardware accelerator (HA).

Fig. 6.8 shows the application profiling results of all the 15 generated SoCs. The figures show the total CPU processing time of the application vs. the FPGA resources (LUT count) utilized for each SoC. The edges indicate the corresponding IP-integration rules, with the starting point as the input SoC and and the end point as the output SoC. Both the bare metal and the Linux OS versions were considered in order to evaluate the effect of HW access latency overhead and data communication overhead on the overall application performances. In the figure, the host SoC architecture node 0 for the bare metal case has more LUTs than the node 0 for the Linux OS case. This is because the host SoC for the Linux OS has camera interface and display units as hard macros in the processing system part of the Zync FPGA.

In both cases, progressive integration of the hardware accelerators reduces the CPU processing time for the application with the increasing cost of additional FPGA resources usage. The figures show the different spread of the performance-area tree for the bare-metal and Linux OS, indicating that the overheads vary for both cases. We can see

Candidate SoCs	CE Processing	ME Processing	EEE Processing	ECE Processing
Candidate 0	CPU	CPU	CPU	CPU
Candidate 1	HA	CPU	CPU	CPU
Candidate 2	CPU	HA	CPU	CPU
Candidate 3	CPU	CPU	HA	CPU
Candidate 4	CPU	CPU	CPU	HA
Candidate 5	HA	HA	CPU	CPU
Candidate 6	HA	CPU	HA	CPU
Candidate 7	HA	CPU	CPU	HA
Candidate 8	CPU	HA	HA	CPU
Candidate 9	CPU	HA	CPU	HA
Candidate 10	CPU	CPU	HA	HA
Candidate 11	HA	HA	HA	CPU
Candidate 12	HA	HA	CPU	HA
Candidate 13	HA	CPU	HA	HA
Candidate 14	CPU	HA	HA	HA
Candidate 15	HA	HA	HA	HA

Table 6.1.: HW-SW mapping of the four tasks of the motion detection application for the host SoC and 15 generated candidate SoCs

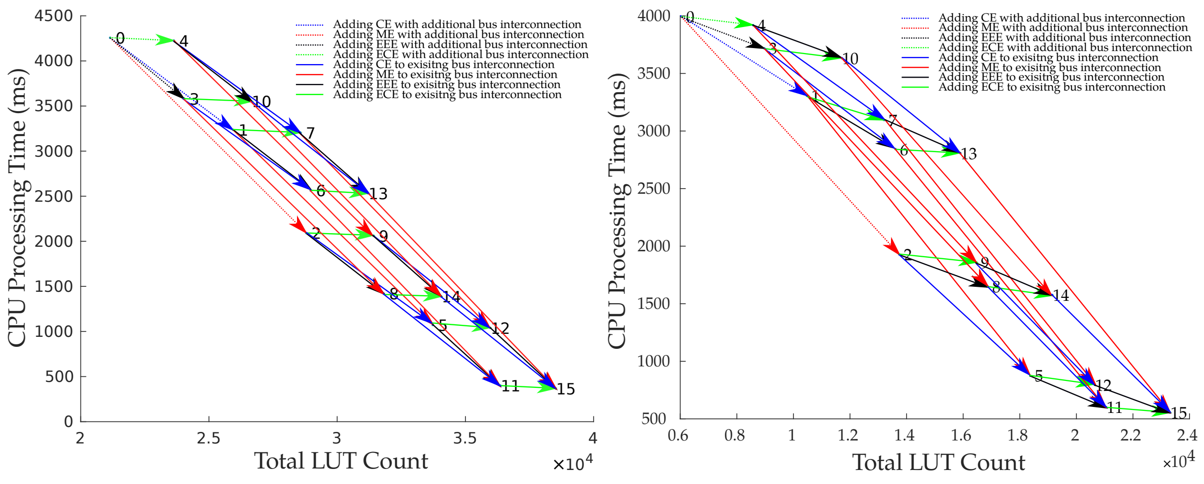


Figure 6.8.: Performance analysis for the application as (a) bare-metal, (b) Linux

a wider spread with the Linux OS mainly because of less performance gains obtained with using CE and EEE HAs. Since the overall HW acceleration of these two HAs is not significant, so the larger OS overheads of data communication and drivers latency is diminishing the overall effectiveness (Table. 6.2).

In the figure, the SoC architectures that are nearer to the origin offer the best performance-area trade-off. These architectures form the *Pareto optimal* front of the performance-area trade-off. For the bare-metal case, Arch. 0, 3, 1, 2, 8, 5, and 11 form the Pareto front, and with the Linux-OS case, Arch. 0, 3, 1, 2, 5, and 11 form the Pareto front. In this case study, both Pareto fronts contain same set of candidate SoCs except the additional

Arch. 8 in the bare-metal case. By comparing the processing times of Arch. 0 and Arch. 15, it can be seen that the frame processing time was reduced for the bare-metal application by a factor of 11x and for the Linux application by 7x. Fig. 6.9 and Fig. 6.10 show the computed motion vectors on the external monitor for the bare metal and the Linux OS based computing respectively.

In this case study, by encoding the SoC DSE knowledge in the IP-integration rules of the IP library, has helped to potentially reduce the couple of days of effort for DSE to a couple of hours using the GRIP tool. The proposed methodology required us to prepare the HASL. It is a one time effort, and at the same time it helps to progressively grow the IP-integration knowledge in the HASL.

	Performance (ms)		**Resource Utilization**	
	Software	**Hardware (HLS)**	**FFs**	**LUTs**
Census Engine	612	102	1812 (1%)	3280 (6%)
Matching Engine	2291	127	1852 (1%)	6539 (12%)
Ego-Motion Estimation	694	21	1700 (1%)	1832 (3%)
Ego-Motion Compensation	41	10	1231 (1%)	1377 (2%)

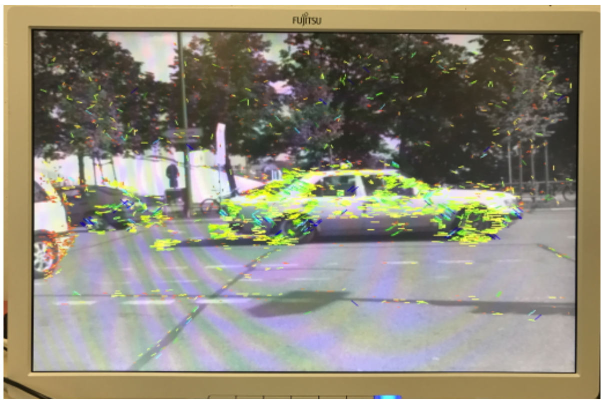
Table 6.2.: Performance comparison (frame processing time) of the motion detection application in the software and hardware implementations

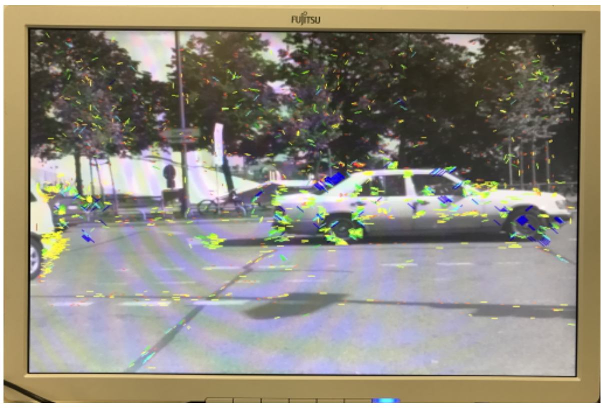
6.5. Case Study 2 - Acceleration of Video Processing Filters

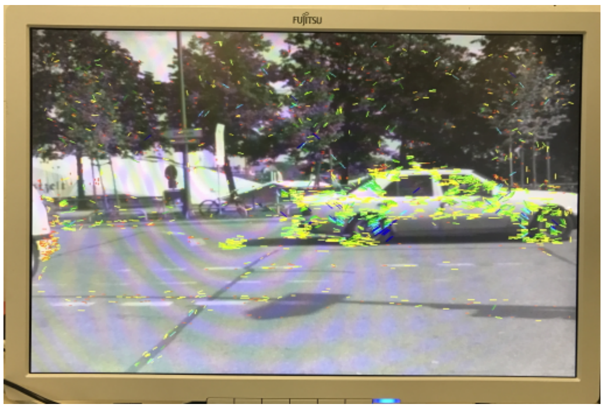
Target Application: The second case study is about the acceleration of a video-processing application on the ZedBoard FPGA using the GRIP tool. This application performs three filtering operations in sequence on an input video stream from an external camera device: Sobel filtering, Erosion filtering and Grayscale conversion. The Sobel filter is an edge detection filter, its iterative gradient operations on 3x3 pixel windows make it computationally expensive. The Erosion filter is also a window-based operation which erodes the brightness of a frame. The Grayscale conversion is a pixel scaling operation, which works pixel-by-pixel. Fig. 6.11 illustrates the data flow for this SW application.

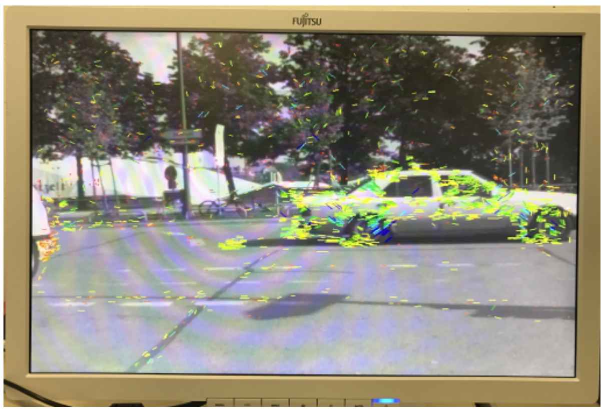
The software application is targeted on the host SoC architecture which is discussed in Sec. 6.2. In the following, we will use that host SoC as a start point to perform DSE for the application using the GRIP approach for both the bare-metal and the Linux-OS based set-ups.

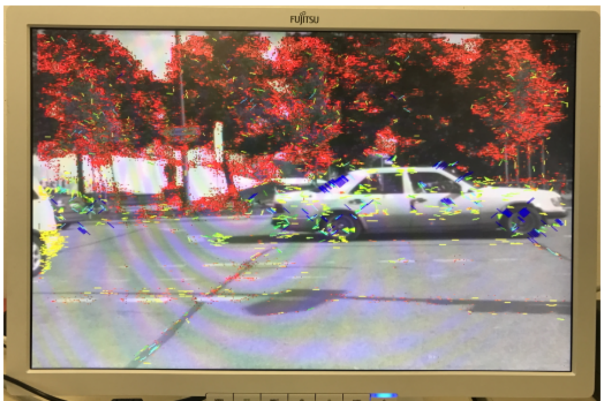
IP Packaging: The IPs required in this case study are: (1) IPs for hardware acceleration for the Sobel filter (SO), Erosion filter (ER) and Grayscale filter (GR) as well as (2) video


(a) Frame 2


(b) Frame 4


(c) Frame 6


(d) Frame 8


(e) Frame 6, non Ego compensated

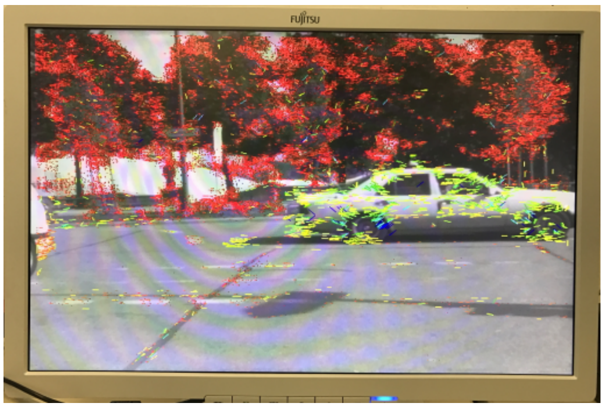
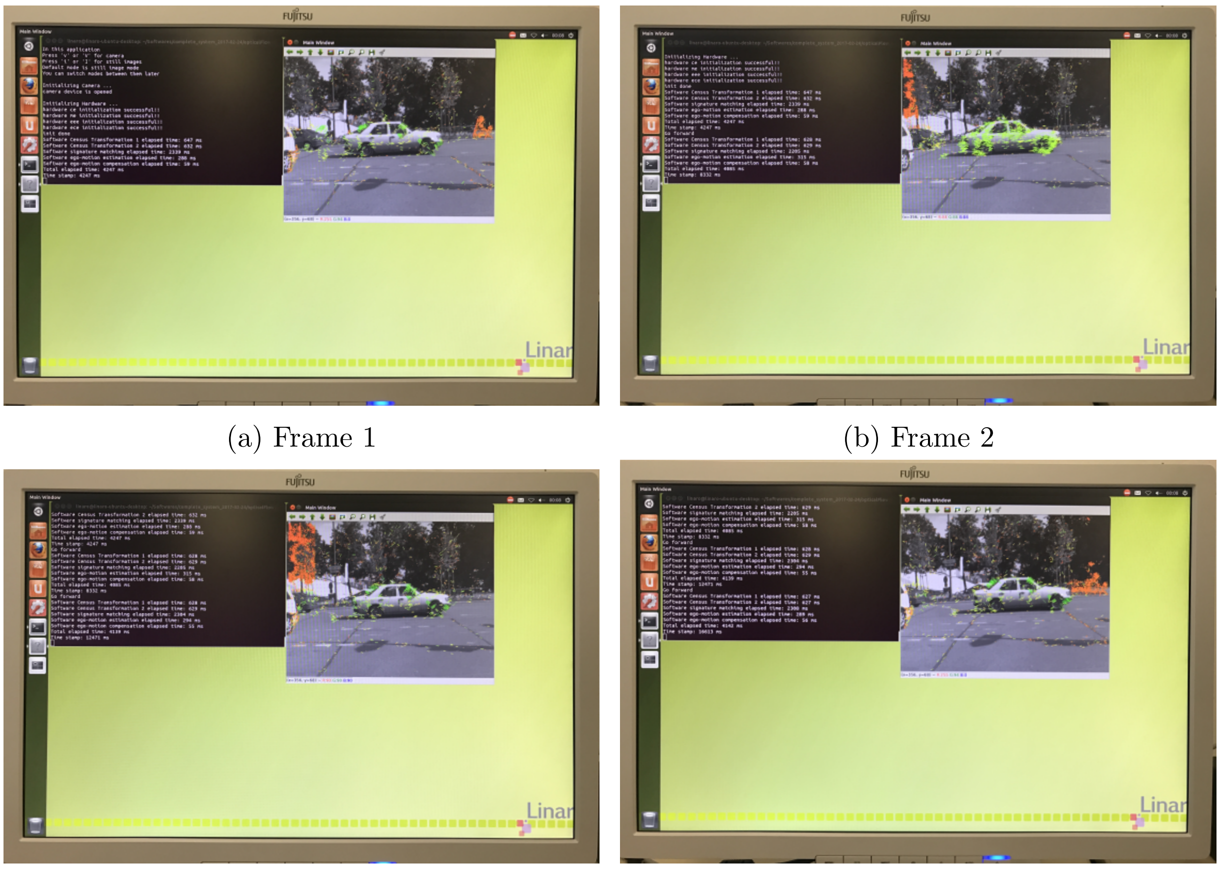

(f) Frame 8, non Ego compensated

Figure 6.9.: Motion detection application with bare metal and SW only implementation

direct memory access (VDMA) for memory communication and (3) other IPs for camera & display interfaces. Each of these functional HW IPs (SO, ER, GR) is associated with three IP-integration rules for integration on a dedicated bus, on an existing bus and in pipeline to an existing HW accelerator IP (See rule 2 to rule 4 in Fig. 3.3). Thus,



(a) Frame 1 (b) Frame 2

(c) Frame 3 (d) Frame 4

Figure 6.10.: Motion detection application on the Linux OS and SW only implementation

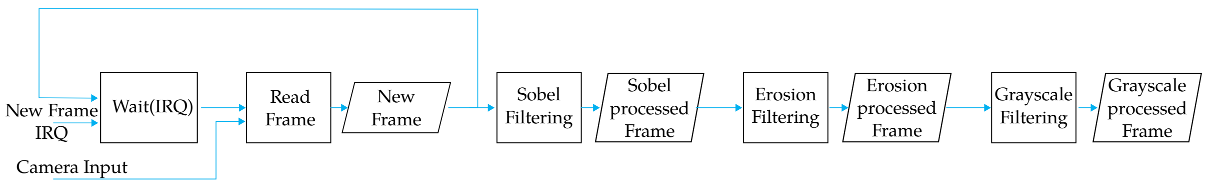


Figure 6.11.: Data flow of the video processing application

in total nine IP-integration rules for all HW IPs. Table 6.3 shows a comparison of the processing times for the SW and HW implementations of these three filtering tasks.

HW-accelerated SW library (HASL): At this step, IP-XACT descriptions for all the required HW IPs and associated IP-integration rule are used to generate the required HW drivers and scheduler. In this case study, an IP subsystem compromises of a VDMA IP connected either to a single HW accelerator IP or to a set of pipelined HW accelerator IPs. Overall, $\sim$ 3000 lines of driver code are generated, while it was required to write $\sim$ 200 lines to customize the drivers to specific IP needs.

	Performance (ms)		Resource Utilization	
	Software	**Hardware (HLS)**	**FFs**	**LUTs**
Sobel Filter	149.2	6.75	980 (1%)	1080 (2%)
Erosion Filter	166.42	6.75	1174 (1%)	1975 (3%)
Grayscale Filter	27.02	6.30	651 (1%)	1007 (2%)
VDMA IP	NA	NA	4176 (4%)	3887 (7%)

Table 6.3.: Performances of various filter tasks executed on CPU and as dedicated HW

Design Space Exploration (DSE): In this DSE, the design space is explored to the depth of three (Fig. 6.12). The node 0 is the host SoC with no accelerator IP, the nodes 1 to 3 have one IP for SO, ER, GR and one VDMA each. At each further depth, more HAs are progressively added to the host SoC. In the figure, the solid edges are indicating parallel integration of HW IPs and the dotted edges indicate IP-integration in pipeline. Fig. 6.13 shows the block diagram for the SoC candidate architecture for node 12. The block diagram shows that the SO and ER filters are operating in pipeline, while ER is integrated in parallel. The DSE tree with depth three has 82 feasible candidate SoCs.

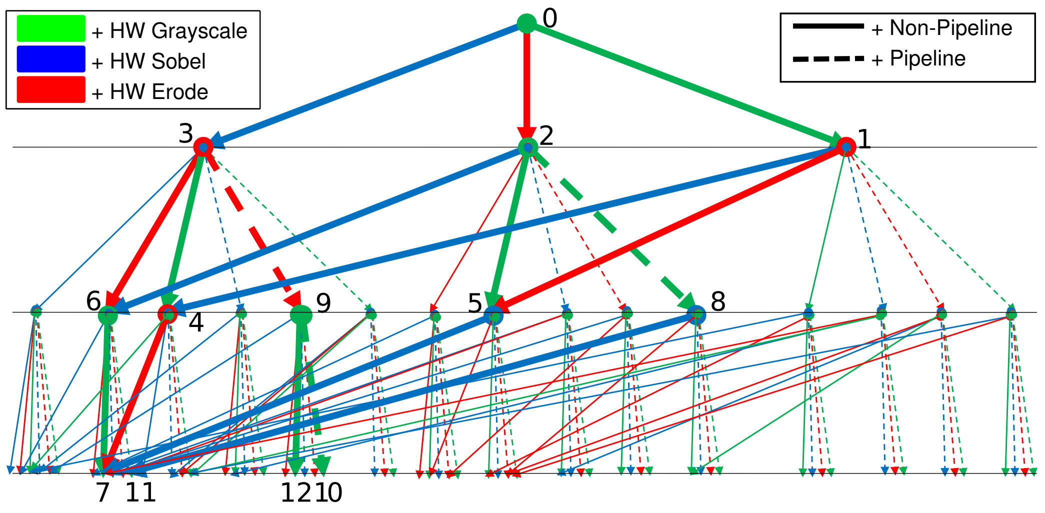


Figure 6.12.: Design space exploration tree for the video processing case study.

DSE Constraints: In this case study, the following DSE constraints are applied to prune the non-optimal SoC architectures from the DSE tree.

1. Allow pipeline integration for SO, ER & GR IP: Application has sequential SW tasks execution on frames without custom operations between the functions. Pipelining reduces communication overhead for writing frames back to memory.

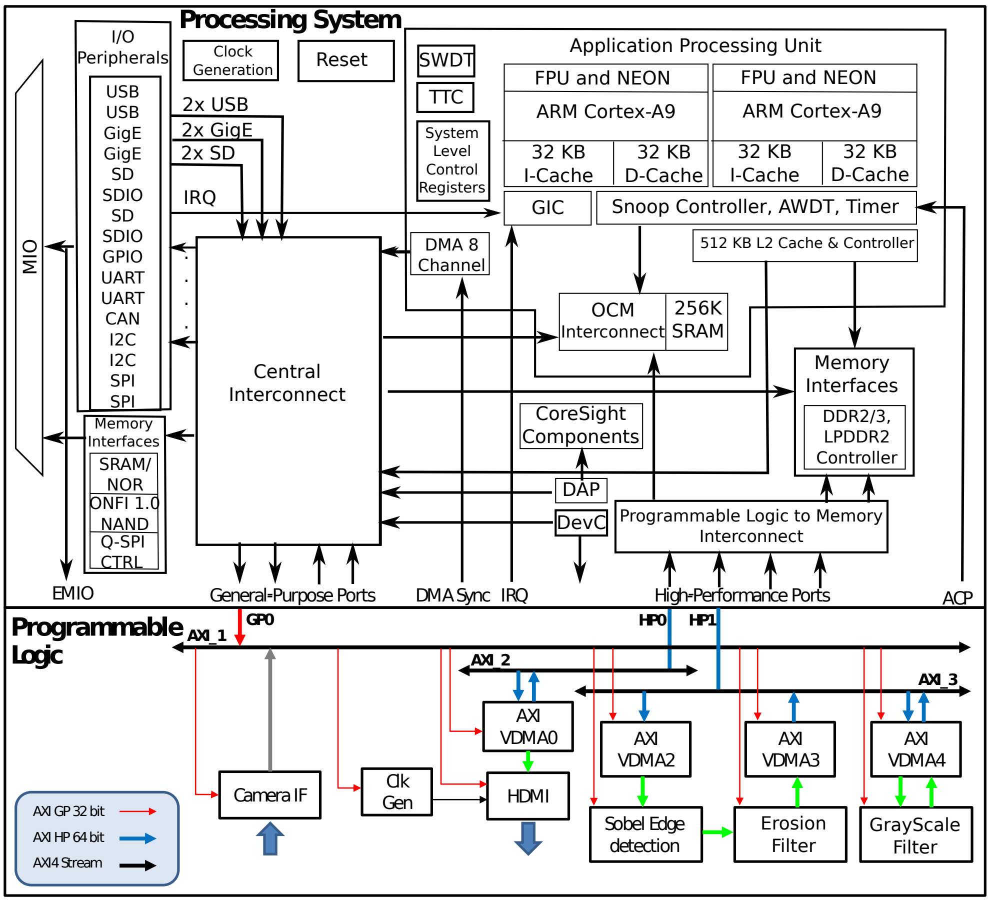


Figure 6.13.: Block-diagram for the video-processing SoC set-up on Zynq FPGA

2. Restrict to single instantiation of each accelerator IP. Since each SW task in the application is sequentially executed once for each frame, the parallel computation will not benefit.

After applying these DSE constraints, the candidate SoCs number is reduced to 12, which are shown as the bold edges and nodes in Fig. 6.12. Table 6.4 describes HW-SW mapping and leveraged pipelining between HAs for these 12 generated SoCs,

Results: Fig. 6.14 shows the performance-area trade-off for implementation as (a) bare-metal application and (b) Linux application. The SoC architectures 0 to 12 correspond to the DSE tree nodes of Fig. 6.12. Arch-0 is the initial SoC design with full SW processing. The bare-metal application accesses the camera inputs via a custom camera interface IP in the FPGA, while the Linux application accesses camera inputs via the USB module in the processing system part of the Zynq chipset. The Y-axis shows the frame processing time (performance) and the X-axis shows the utilized FPGA resources (area) for all SoC candidates. For the bare-metal application, the integration of any

Candidate SoCs	SO Processing	Pipe	ER Processing	Pipe	GR Processing
Candidate 0	CPU		CPU		CPU
Candidate 1	CPU		CPU		HA
Candidate 2	CPU		HA		CPU
Candidate 3	HA		CPU		CPU
Candidate 4	HA		CPU		HA
Candidate 5	CPU		HA		HA
Candidate 6	HA		HA		CPU
Candidate 7	HA		HA		HA
Candidate 8	CPU		HA	YES	HA
Candidate 9	HA	YES	HA		CPU
Candidate 10	HA	YES	HA	YES	HA
Candidate 11	HA		HA	YES	HA
Candidate 12	HA	YES	HA		HA

Table 6.4.: HW-SW mapping of tasks of the video processing application for 12 generated SoCs. 'YES' in Pipe column indicates pipelining among adjacent HAs

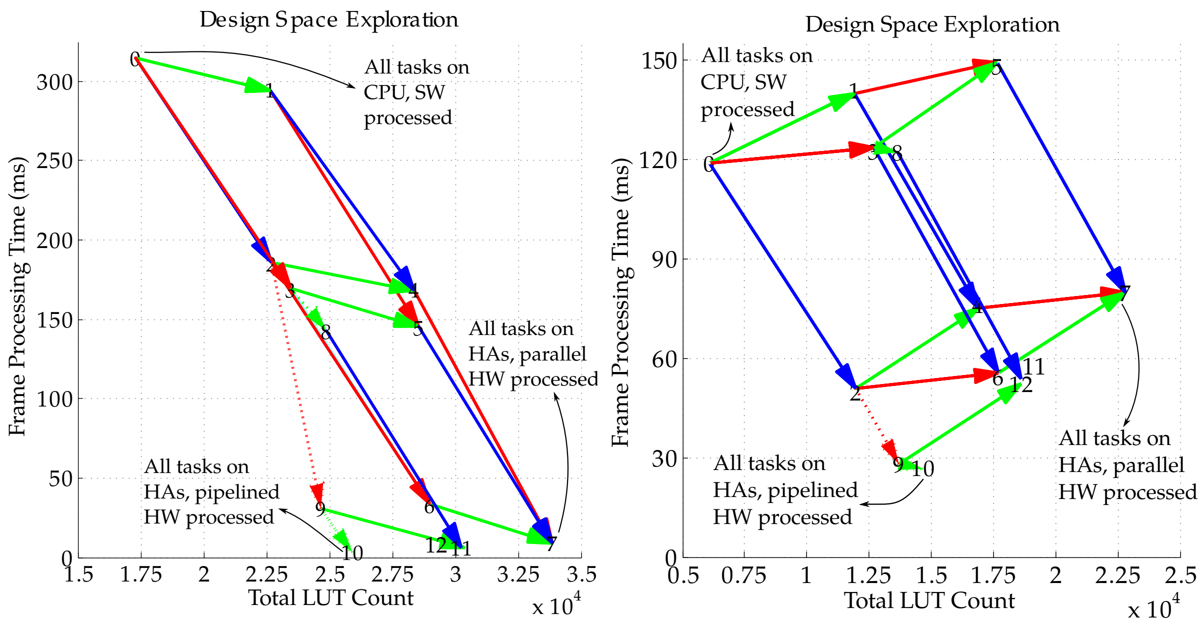


Figure 6.14.: Performance analysis for the application as (a) bare-metal, (b) Linux

accelerator IP yields performance improvement. Most improvement is gained by the SO IP. Pipelining brings some additional improvement.

For the Linux based application, non-pipelined integration of ER & GR filters lead to worse (longer) frame processing time. This is caused by larger communication overheads for accessing hardware accelerators with the OS-layer. The system memory of the Linux-OS is inaccessible for the hardware accelerators and the CPU must transfer the video data from the system memory to the non-system memory for hardware processing, which requires additional processing cycles. When ER and GR are integrated in pipeline, the overhead is removed and some benefit can be observed. This shows the importance of performing the design estimations at the FPGA prototyping level, as the communication loads can be accurately observed. In the performance curves, for non-OS, the architectures 0, 2, 3, 9, and 10 form the Pareto front, while the architectures 0, 2, 9, and 10 form the Pareto front for the Linux-based case. Frame processing time was reduced for the bare-metal application by a factor of 150x and for the Linux application by 4x.

In this case study, an additional analysis is performed on the performance overhead of the generated HW drivers w.r.t. manually developed HW drivers. Fig. 6.15 shows the overhead of accessing the IP subsystems for all 12 candidate SoCs. It was observed that the generated HW drivers are up to 3x slower than manual HW drivers. However, CPU cycles used by HW drivers are only a few thousand clock cycles compared to couple of million of clock cycles for application processing. For this case study, the worst-case overhead for the generated HW drivers was found to less than 0.28% of the overall application processing.

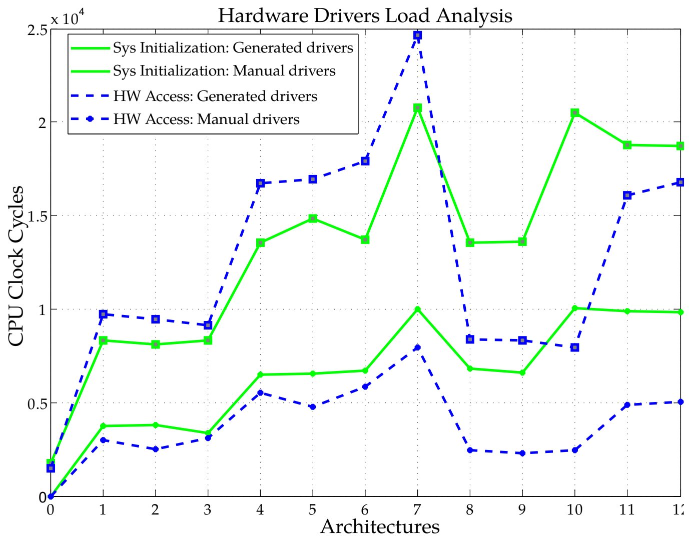


Figure 6.15.: HW access overhead for the generated and manually written flat drivers.

6.6. Conclusions

In this chapter, we discussed two computer vision applications as independent case studies: a motion detection application; a sequential video processing filtering application. Using the proposed approach in the previous chapters, a SW-defined design space exploration is performed on both of these applications using the GRIP tool. We discussed the preparation steps of the required hardware-accelerated software libraries (HASL) and the IP-integration rules. During the DSE, progressively the computationally intensive SW tasks are moved from the targeted ARM CPU to the respective HW accelerators.

Through the DSE of these case studies, we achieved the Pareto optimal candidate architecture for the performance-area trade-off, both for the bare metal and the Linux OS based implementations. The different SoC candidates on the Pareto optimal fronts for the bare metal and Linux OS confirm the varying influences on the performance-area trade-off of the data communication within the HW-drivers layer.

In these case studies, the application performances are improved by a factor of 10x-150x for the bare metal, and a factor of 4x-7x for the Linux OS during the DSE.

7. The GRIP Tool with Feedback - SoC Performance Monitoring

"...major FPGA vendors today have internal logic analyzers to address the visibility issue [signals probing inside FPGA]. However, many of these internal logic analyzers have several limitations, including support for only single FPGA debug, limited memory size using FPGA internal memory, and long place-and-route times to change probes.", Ron Green, Technical Communications Manager, S2C Inc.

7.1. Introduction and Problem

Accurate system performance estimation on an FPGA is important for SoC architecture refinements. Typically, for the performance analysis of the SoCs, the FPGA vendors provide multiple hardware monitoring and debugging IPs. Analyzing an SoC FPGA prototype requires the correct integration of additional monitoring and debugging IPs into an SoC. Integration of IPs into an SoC is a challenging task as it requires additional HW knowledge on these IPs. In addition, under the resource constraints of the FPGA, not all desired hardware monitors (HM) may fit into the FPGA together with the SoC under investigation. The set of desired analysis tasks is split into multiple subsets, and the required HMs in each subset must fit into the FGPA. This splitting must be done in a way to minimize the number of these subsets. Fig. 7.1 shows the integration of an AXI-bus monitor IP in an SoC.

In this chapter, we are automating the process of integrating HMs into SoCs using the GRIP tool on resource-constrained FPGAs. With this automation we are making the FPGA bring up process faster and less error-prone than the available ad-hoc methods. Our work solves this problem in a two step approach. As first step, we prepare the IP-integration rules for HMs in the GRIP tool , and then formulate the problem of HMs integration as the bin packing problem. A bin packing algorithm is used to generate an extensive set of SoC monitoring architectures required for complete analysis, constrained to the target FPGA resources. The algorithm uses the estimated resource utilization as the feedback in this automation. The GRIP tool generates the required FPGA design description files of the SoCs with integrated HMs to be taken further for HW synthesis.

The main contribution of this chapter is the bin-packing problem formulation for generating SoCs for analysis under FPGA resource constraints, within the GRIP tool.

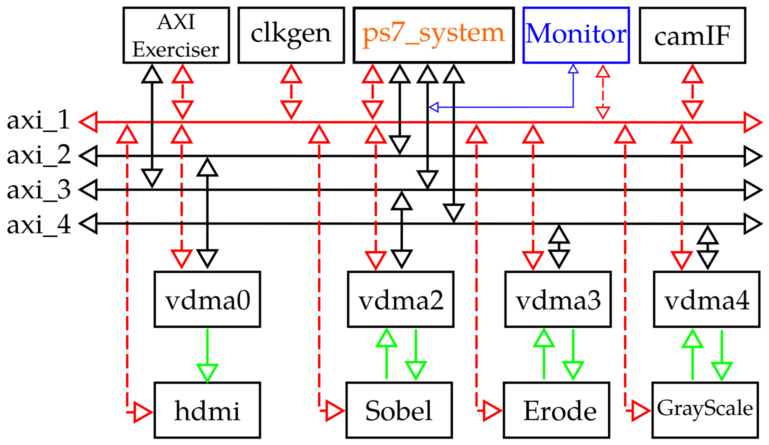


Figure 7.1.: Debugging and performance monitoring on AXI-bus on the ZedBoard

7.2. Problem Formulation - Bin Packing

The GRIP platform provides powerful utilities for the IP-based SoC design automation. Previously, we have discussed using the GRIP tool progressive design space exploration. In that approach, the DSE constraints extracted from the data flow of the target SW application are utilized to guide the DSE. The approach is sequential, and has the advantage that without the design performance estimations, it can quickly generate a set of candidate SoCs.

In this chapter, the GRIP tool is utilized for automating SoC performance monitoring and debugging on an FPGA. Each monitoring task requires a set of additional hardware monitors (HMs) to be integrated at specific positions in an investigated SoC. For the resource-constrained FPGAs, the complete set of monitoring or debugging tasks is to be split into multiple subsets, which form the *partition* of the original set. The partition of the original set means that all the subsets are non-empty and mutually disjoint, and their union forms the original set. The splitting of original set into the subsets is constrained to the available FPGA resources, i.e. the total FPGA hardware resources required to perform analysis tasks defined by each subset must not exceed the available FPGA resources.

We propose to formulate this as a bin-packing problem. The algorithm uses the estimation of the FPGA resources while bounding them to the maximum available resources. Here is the formulation of the problem,

$$
\begin{aligned}
A &= \{x : \ x \text{ is a desired analysis task}\} &\quad (7.1)\\
P &= \{Y : Y \subseteq A\} &\quad (7.2)
\end{aligned}
$$

$$
\begin{array}{c} \text{s.t. } P \\ \text{is a partition} \end{array} \quad : \quad \begin{cases} \forall_{Y \in P}\,(Y \neq \emptyset) \\ \forall_{X,Y \in P}\,(X \neq Y \rightarrow X \cap Y = \emptyset) \\ (\cup_{Y \in P} Y) = A \end{cases} \qquad (7.3)
$$

$$
\text{and, with} \qquad \forall_{Y \in P}\,(\mathcal{F}(Y) \leq C) \qquad (7.4)
$$

The problem statement is a classical bin packing problem (formulated as equations 7.1-7.4), which is computationally a NP-hard problem. For the SoC analysis problem, set A contains all the desired analysis tasks and C is a set of the FPGA-resource constraints. The complete set of FPGA monitoring tasks is split into set of subsets of the analysis tasks P, where each element $(Y \in P) \subseteq A$ obeys the FPGA resource constraints. Function $\mathcal{F}$ estimates the resource utilization of the modified SoC architecture on the desired FPGA, and this must fit to the resource constraints vector C. The set P is a partition of the set A, the union of tasks of each element Y of set P covers the complete space of the set A.

Algorithm 3: Given an input set A of all required analysis tasks, a vector C of maximum available FPGA resources, the algorithm returns the set P of the subsets of the analysis tasks, each element of which obeys the FPGA resource constraints C

```
Input  : $A$: analysis tasks set;
         $C$: FPGA resource constraints set
Output: $P$: set of subsets of monitoring tasks
Reduce ($\mathcal{A}$, $\mathcal{C}$):
$N = \emptyset$;
if $\mathcal{F}(\mathcal{A}) \leq \mathcal{C}$ then
    return $\mathcal{A}$;
else
    foreach $x \in \mathcal{A}$ do
        $\mathcal{A} = \mathcal{A}/\{x\}$;
        $N = N \cup \{x\}$;
        if $\mathcal{F}(\mathcal{A}) \leq \mathcal{C}$ then
            return $\{\mathcal{A}, \text{Reduce}(N, \mathcal{C})\}$
        end
    end
end
P = Reduce (A, C);
return P;
```

Algo. 3 is used to solve this problem, it generates a partition P of A as defined by equation (7.3), which has the *minimal cardinality*, s.t. each element of P obeys the constraints of equation 7.4. The algorithm recursively uses the function *Reduce(A,C)* to reduce the set A into the set of subsets, P. The set A of all the desired analysis tasks is defined as a set of IP-integration rules (already available in the IP library). Each rule corresponds to the integration of a single HM IP. The algorithm then generates the partition set P of elements $Y : Y \subseteq A$ using the Algo. 3.

The IP-integration rules in each subset Y contain the integration knowledge for required HW IPs to be added to the SoC under consideration (host SoC) for the desired monitoring or debugging tasks. In the next section, we will discuss the details of using this algorithm with the GRIP tool. The GRIP tool uses the structural information of Y.

7.3. Integration with the GRIP tool

Fig. 7.2 shows the integration of the FPGA resource estimation feedback with the GRIP tool. As discussed in the previous section, the feedback is used in the bin packing algorithm. The GRIP tool generates the new SoC candidates with the integrated HMs. This is achieved by few *vendor extensions* in the IP-XACT component descriptions. Each IP-XACT component description in the IP library is appended with additional information on their respective estimated FPGA resource usages. These *vendor extensions* are used with the IP-XACT design description of the generated candidate SoCs to estimate the required FPGA resources. Later, the estimated SoC resources are compared to the maximum available FPGA resources, which is available in an FPGA resource constraints file. Fig. 7.3 shows the FPGA constraints file for the Xilinx Zynq chipset.

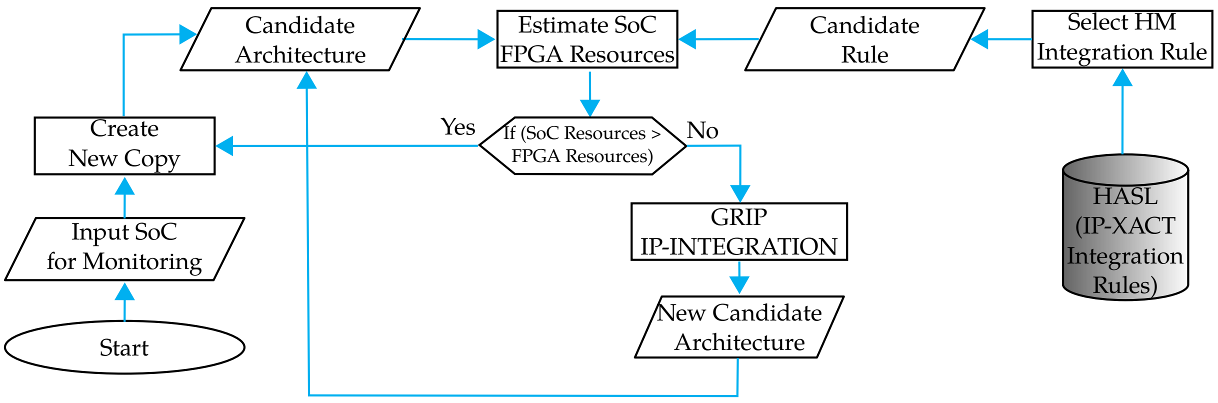

Candidate Architecture
Estimate SoC FPGA Resources
Candidate Rule
Select HM Integration Rule
Create New Copy
Yes
If (SoC Resources > FPGA Resources)
No
GRIP IP-INTEGRATION
HASL (IP-XACT Integration Rules)
Input SoC for Monitoring
New Candidate Architecture
Start

Figure 7.2.: GRIP tool with the FPGA resources estimation feedback for SoC monitoring

The set of analysis tasks is defined by the user by listing already available IP-integration rules in the IP library. The *GRIP engine* then generates the set of subsets of IP-integration rules that are compliant to the FPGA constraints (the set P for the Algo.

```
<?xml version="1.0" encoding="ASCII"?>
<fpga:FPGAconstraints>
     <fpga:fpgaSynopsis/>
     <fpga:fpgaXilinx>
         <fpga:product>Zynq-7000</w3schools:product>
         <fpga:family>Zynq-7000</w3schools:family>
         <fpga:package>cg1484</w3schools:package>
         <fpga:speed>-1</w3schools:speed>
         <fpga:resourceconstraints>
              <fpga:maxLUT>53200</w3schools:maxLUT>
              <fpga:maxFF>106400</w3schools:maxFF>
              <fpga:maxBRAM>560</w3schools:maxBRAM>
              <fpga:maxDSP>200</w3schools:maxDSP>
              <fpga:maxLogic>85000</w3schools:maxLogic>
         </fpga:resourceconstraints>
     </fpga:fpgaXilinx>
</fpga:FPGAconstraints>
```

Figure 7.3.: FPGA resource constraints defined in an XML constraints file

3). The set of IP-integration rules in Y are then *applied* to the host SoC, to generate the IP-XACT description for new SoCs corresponding to each set Y. The *code generation engine* generates all the required design files for the new SoCs that are then programed on an FPGA. In the current implementation we support the FPGA design files generation for the targeted Xilinx-Zynq FPGA chipset.

The methodology as described above is used when an extensive set of analysis tasks on an SoC is desired. In another mode, the system architect can also step-by-step guide the SoC generation for specific and limited SoC analysis (especially for the debugging purposes, where limited number of debugging IPs are required to be integrated to the SoC).

7.4. Case Study: Bus-Load Monitoring of a Video-Processing SoC

In this case study, the developed performance analysis partitioning algorithm is used to perform bus-load analysis for a candidate SoC obtained for the video processing case study 2 from Chapter 6. The architecture of the SoC is as shown in Fig. 7.4. The SoC contains four bus systems: `AXI_1: AXI LITE; AXI_2, AXI_2, AXI_3: AXI4`. The `AXI_1` is a slow speed bus, which transfers video frames from the external camera device to the DDR memory. `AXI_2, AXI_3, and AXI_4` are the high speed buses for handling the image data transfers to-and-from the hardware accelerators. The bus load analysis is important to understand the data communication conflicts and their impact on the overall system performance [1].

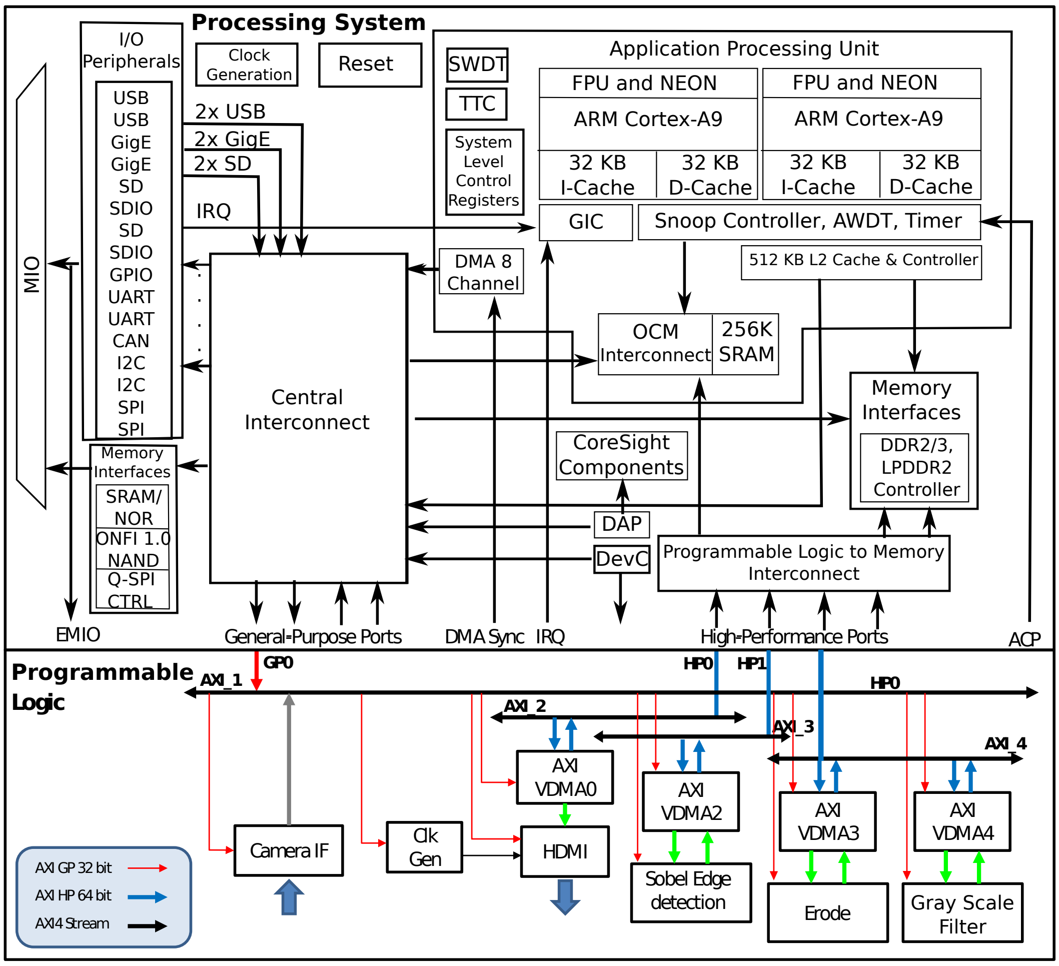


Figure 7.4.: Video processing SoC setup on ZedBoard with three hardware accelerators

For the extensive performance analysis of this system the Xilinx *AXI-Monitor* IP (HW monitor, HM) is used to monitor all the AXI4 system buses (`AXI_2, AXI_3, and AXI_4`). AXI-Monitor IP is one of the performance monitoring and debugging IP to analyze the Read/Write transactions and latencies of the data accesses from the Masters modules in the programmable logic (PL). The complete analysis of all three AXI4 buses required to insert three AXI-Monitor IPs, one on each AXI4 bus. However, insertion of all three HMs overshoots the available FPGA resources.

For this simple case study, the IP-integration knowledge of the AXI-monitor IP is encoded by the IP-integration rule (Fig. 7.5). Since the bus-load analyses on these three

[1]In this case study, the off-chip data communication overhead is much more dominant than the on-chip data communication conflicts. This is also seen in the results presented in Chapter 6 as the varying Pareto front of the performance-area trade-offs for the Bare metal and with Linux OS.

buses are independent, a single rule can describe the required changes. In the case of dependent analysis, the dependency is captured by the context among the LHS and RHS patterns of the rule. After defining the problem using the HM-integration rule, the GRIP tool generated two IP-XACT SoC architectures for the extensive performance analysis [2].

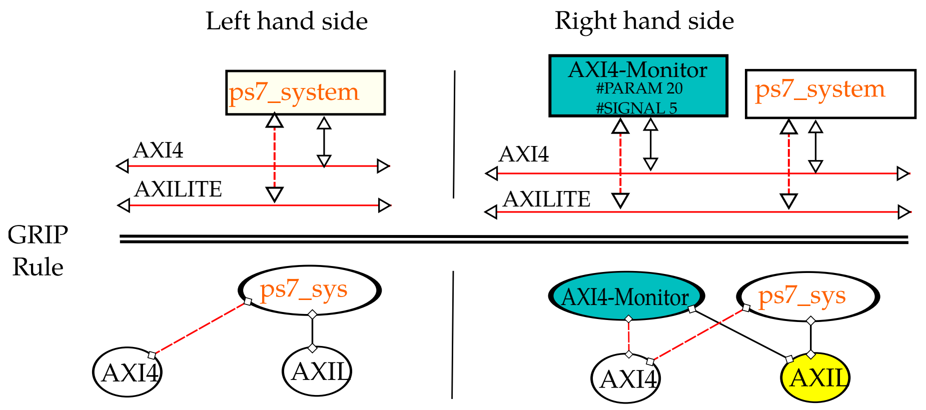


Figure 7.5.: IP-integration rule to integrate an AXI-monitor IP to an AXI4 bus

For step-by-step interaction mode, the same hardware system was evaluated for the target of analyzing the system performance when the `AXI_3` bus is fully utilized. An *AXI-Exerciser* IP is used to overload the bus and an *AXI-Monitor* IP to analyze the bus load. The strategy for load analysis is as defined below using the available tool functions and shown in a simplified block diagram in Fig. 7.1.

```
addExerciser = GenerateSpecificRule(<HA Exerciser>, <AXI4,axi_3>)
addMonitor = GenerateSpecificRule(<HA Monitor>, <AXI4,axi_3>)
newAGR = ApplyRule(addExerciser, hostAGR)
newAGR = ApplyRule(addMonitor, newAGR)
ExportIPXACT(newAGR, <outDir>)
```

In the code above, we use function `GenerateSpecificRule(<args>)` to generate IP-integration rules for integrating *Exerciser* IP and *Monitor* IP. The function takes arguments as a desired HA and a bus instance name as the location of IP integration to generate an IP-integration rule. After implementing the desired strategy, the tool executed all the steps defined by the strategy. At completion, the tool generated IP-XACT SoC architecture descriptions and the FPGA design configuration files to be taken forward for synthesis on the ZedBoard.

[2]The results of SoC on-chip bus-load analysis are not relevant in the scope of this chapter, hence are not presented.

7.5. Conclusions

In this chapter, we tried to solve the problem of SoC performance monitoring (using HW monitors) with the FPGA resource constraints. Under the FPGA resource constraints, the insertion of HW monitors might be required to split into multiple SoC architectures. The problem is formulated as a multi-dimension bin-packing problem, with the objective to obtain a set of candidate SoCs with minimum cardinality. So, the objective is to obtain minimum candidate SoCs that are sufficient to perform all required performance monitoring tasks.

This chapter extends the GRIP tool to include the HW resources estimation as feedback, while performing the iterative rules application. For this, the GRIP IP integration engine is extended to estimate the required FPGA resources for the generated candidate SoCs, and eventually generating only those candidate SoCs that are implementable on the Zynq FPGA.

This approach is demonstrated on a bus-load analysis case study, where the GRIP tool generates the set of synthesizable SoCs covering the complete bus-load analysis set.

8. Conclusions

The objective of the presented work is to bridge the knowledge gaps among the two actors in the IP-based SoC designs: the IP Supplier, and the SoC architect. The goal is to further strengthen the increasing IP reuse for the IP-based SoC designs. More precisely this work dealt with the challenges of the IP exchange. An objective for the SoC architect is to quickly explore the design space to obtain the Pareto optimal SoC architectures, which offer best performance-cost trade-offs. This work tries to enable using the third-party IP library for the SoC design as easy as using the software library for software application development.

The software-defined SoC design is one special case of the IP-based SoC design methodologies. Here, the SoC design information is encoded in the target software application description. The computationally intensive tasks of the target application are progressively transferred to the dedicated hardware (HW) IP subsystems available in the hardware IP library. Another objective of this work is to solve the challenges of the software-defined SoC design of iteratively introducing new HW IP subsystems into an SoC and adapting the software drivers accordingly.

The target hardware platform for SoC prototyping of choice is the FPGA-based development board, the Xilinx ZedBoard. One intent that is followed in this work is to enable the SoC design automation targeted to generate synthesizable HW descriptions. So, the objective of all the automation algorithms developed is to bring-up the generated system on the FPGA development board. This objective deals with generating HW and SW descriptions, and seamless transfer of controls among various HW processing units.

The proposed solutions allow the possibility of efficient design automation for HW synthesis together with HW drivers generation for SW. The solution utilizes the graphs and principles of graph grammars to formulate structural information of the SoCs. It uses rule-based graph rewriting to encode and automate the structure transformations. The domain-specific SoC descriptions are represented using the IEEE-1685-2009 IP-XACT standard. In the graph space, an SoC is represented using the architectural graph representation (AGR). Various model-to-model transformation engines are developed to transparently transform the SoC descriptions among the domain-specific space and the graph space. An IP-XACT rule encodes the desired SoC structural changes. The rule consists of LHS and RHS IP-XACT design patterns. The IP-XACT IP components together with the IP-integration rules form the IP package.

In order to adapt the target software application to the modified SoC, we propose to generate the HW-access drivers from the IP-XACT descriptions during the library preparation. The HW drivers are generated by the *GRIP HASL generation engine*, which generates the drivers for both the bare metal and the Linux OS based application development. The drivers also have a simple scheduler that takes care of executing a software application task either on the Master CPU or the dedicated HW-accelerating sub-system of the SoC. The hardware-accelerated software library (HASL) so prepared contains the IP-XACT IPs, corresponding IP-integration rules, and the HW-access drivers. In this way, the HASL bridges the HW knowledge gap of the software developer.

During the SoC structural transformations, the IP-XACT rules are first transformed to corresponding graph representations and then applied to a host SoC design to attain a structural change. All the structural changes are performed in the graph space using the *GRIP IP-integration engine*. After the structural changes, the SoC is transformed back to the domain-specific IP-XACT space. Here, the SoC is verified using the *GRIP design verification engine* and is then targeted to the desired HW platform using the *GRIP code generation engine*.

Once the automation for both the software (HW-access drivers) and hardware generation is available, the iterative application of the HASL IP-integration rules explores the SoC design space. The candidate IP-XACT SoCs obtained by the *GRIP design space exploration (DSE) engine* are transformed to the HW descriptions for the target HW platform using the *GRIP code generation engine*. In this work, the code generation engine supports the system bring-up on the Xilinx Zynq FPGA chipset. It is observed that the number of SoC candidates during the DSE of a software-defined SoC grows exponentially w.r.t the available IP-integration rules in the target HASL. This work introduces DSE constraints to prune the non-desirable SoCs, and to quickly attain the Pareto optimal SoCs for the performance-cost trade-offs.

The proposed methodology and the GRIP tool are demonstrated on two computer vision application case studies. In these case studies, the GRIP tool is utilized to accelerate the target software applications by performing the SoC DSE on a domain-specific HASL. As result, we obtained designs on the Pareto optimal front for both the bare-metal and with the Linux OS. In the case studies, the non-optimal SoC candidates are pruned using the DSE constraints extracted from the data flow of the application. In these case studies, the application performances are improved by a factor of 10x-150x for the bare metal, and a factor of 4x-7x for the Linux OS during the DSE. This design automation reduces the time for DSE from a couple of days, when performed manually or partly automated, to few hours using the GRIP tool.

Bibliography

Accelera (2009): Accelera website, http://www.accellera.org.

AMBA Specifications (2012): https://www.arm.com/products/system-ip/amba-specifications.

André, C, Mallet, F, Mehmood, A & de Simone, R (2008): Modeling SPIRIT IP-XACT with UML MARTE, Proceedings of the DATE workshop on modeling and analysis of real-time and embedded systems with the MARTE UML profile.

Apvrille, L., Muhammad, W., Ameur-Boulifa, R., Coudert, S. & Pacalet, R. (2006): A UML-based Environment for System Design Space Exploration, 2006 13th IEEE International Conference on Electronics, Circuits and Systems.

ARM (2016): ARM Cortex-A9, Technical Reference Manual (revision r4p1).

Ascia, G., Catania, V. & Palesi, M. (2004): A GA-Based Design Space Exploration Framework for Parameterized System on-a-Chip Platforms, IEEE Trans. Evolutionary Computation.

Aulagnier, D., Koudri, A, Lecomte, S, Soulard, P, Champeau, J, Vidal, J, Perrouin, G & Leray, P (2009): SoC/SoPC Development Using MDD and MARTE Profile.

Balarin, F. (2001): Metropolis: A Design Environment for Heterogeneous Systems, Cadence Tech. Conference.

Beltrame, G., Fossati, L. & Sciuto, D. (2010): Decision-theoretic design space exploration of multiprocessor platforms, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

Bhattacharya, A., Konar, A., Das, S., Grosan, C. & Abraham, A. (2010): Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm, International Conference on Complex, Intelligent and Software Intensive Systems.

Booggie tool (2009): http://www.booggie.org.

Botella, P., Sánchez, P. & Posadas, H. (2010): Automatic Generation of SystemC SMP Models for HW/SW Co-Simulation", Conf. on Design of Circuits and Integrated Systems (DCIS).

Brooks, C. (2005): Heterogeneous Concurrent Modeling and Design in Java (Vol. 1: Introduction to Ptolemy II), Tech. report UCB/ERL M05/21, Univ. of California, Berkeley.

Buchmann, T., Westfechtel, B. & Winetzhammer, S. (2012): The Added Value of Programmed Graph Transformations, A Case Study from Software Configuration Management, Applications of Graph Transformations with Industrial Relevance, Lecture Notes in Computer Science.

Chen, H., Godet-Bar, G., Rousseau, F. & Petrot, F. (2014): Device Driver Generation Targeting Multiple Operating Systems Using a Model-driven Methodology, Rapid System Prototyping (RSP).

Chou, P.H., Ortega, R.B. & Borriello, G. (1995): The Chinook Hardware/Software Co-Synthesis System, International Symposium on System Synthesis.

Coffman, EG (2013): Bin packing approximation algorithms: survey and classification, Handbook of Combinatorial Optimization. Springer New York.

Corbet, J., Rubini, A. & Kroah-Hartman, G. (2005): LINUX DEVICE DRIVERS, Third Edition, O'Reilly.

Cordella, L. P., Foggia, P., Sansone, C. & Vento, M. (2001): An Improved Algorithm for Matching Large Graphs, Proceedings of the 3rd IAPR TC-15 Workshop on Graph-Based Representations in Pattern Recognition.

Denney, E. & Trac, S. (2008): A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code, Proceedings of Aerospace Conference,IEEE.

Densmore, D., Donlin, A. & Sangiovanni-Vincentelli, A. (2006): FPGA Architecture Characterization for System Level Performance Analysis, Design, Automation, and Testing in Europe, DATE.

Ecker, W., Esen, V., Nageldinger, U, Steininger, T. & Velten, M. (2008): UML based Code Generation for the HW/SW Interface, I5th International UML for SoC Design Workshop, DAC.

Ecker, W., Mueller, W. & Doemer, R. (2009): Hardware-dependent Software: Principles and Practice, Springer Science and Business Media.

Eclipse Modelling Framework (EMF) (2011): www.eclipse.org/modeling/emf/.

Ehrig, H., Kreowski, H. J., Maggiolo-Schettini, A., Rosen, B. K. & Winkowski, J. (1981): TTransformations of Structures - An Algebraic Approach, Math. Syst. Theory, Lecture Notes in Computer Science.

eMoflon (n.d.): http://www.moflon.org/.

Epsilon (2006): http://www.eclipse.org/epsilon/.

Epsilon - EOL (2006): http://www.eclipse.org/epsilon/doc/eol/.

Epsilon - EVL (2006): http://www.eclipse.org/epsilon/doc/evl/.

Ferrandi, F., Lanzi, P. L., Pilato, C., Sciuto, D. & Tumeo, A. (2010): Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems, Computer-Aided Design of Integrated Circuits and Systems.

Gajski, D. D. & Kuhn, R. H. (1983): Guest Editors' Introduction: New VLSI Tools, Computer.

Geiss, R., Batz, G. V., Grund, D., Hack, S. & Szalkowski, A. (2006): GrGen: A Fast SPO Based Graph Rewriting Tool, Graph Transformations, Springer, Berlin.

Gérard, S., Terrier, F. & Tanguy, Y. (2002): Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML, Springer Berlin Heidelberg, Berlin, Heidelberg.

Givargis, T. & Vahid, F. (2002): Platune: A Tuning Framework for System-on-a-Chip Platforms, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.

GrGen tool v3.0 (20011): http://www.info.uni-karlsruhe.de/software/grgen/.

Gries, M. (2003): Methods for Evaluating and Covering the Design Space during Early Design Development, VLSI Journal.

Guo, X., Chen, Z. & Schaumont, P. (2008): Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors, Embedded Computer Systems: Architectures, Modeling, and Simulation.

Herrera, F., Posadas, H., Villar, E. & Calvo, D. (2012): Enhanced IP-XACT Platform Descriptions for Automatic Generation from UML-MARTE of Fast Performance Models for DSE, Digital System Design (DSD).

IEEE (2009): IEEE 1685-2009 IP-XACT, http://standards.ieee.org.

IEEE (2014): IEEE 1685-2014 IP-XACT, http://standards.ieee.org.

ISO (2004): ISO/IEC 19501:2005 Unified Modeling Language (UML) Version 1.4.2, International Organization for Standardization.

ITRS (2011): International Technology Roadmap for Semiconductors, www.itrs.net, http://www.itrs.net.

Jassi, M., Bordes, B., Mueller-Gritschneder, D. & Schlichtmann, U. (2015): Automation of FPGA Performance Monitoring and Debugging Using IP-XACT and Graph-Grammars, 2015 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD).

Jassi, M., Hu, Y., Mueller-Gritschneder, D. & Schlichtmann, U. (2018): GRIP - Graph-Grammar-Based IP-Integration - An EDA Tool for Software-Defined SoCs, ACM Trans. Des. Autom. Electron. Syst., ACM, New York, NY, USA.

Jassi, M., Mueller-Gritschneder, D. & Schlichtmann, U. (2015): GRIP: Grammar-Based IP Integration and Packaging for Acceleration-Rich SoC Designs, Proceedings of Design Automation Conference (DAC).

Jassi, M., Sharif, U., Mueller-Gritschneder, D. & Schlichtmann, U. (2016): Hardware-Accelerated Software Libraries Drivers Generation for IP-Centric SoC Designs, 2016 International Great Lakes Symposium on VLSI (GLSVLSI).

Java - FreeMarker (2015): ver. 2.3.24, http://www.http://freemarker.org/.

Kactus2 tool (2012): http://www.funbase.cs.tut.fi.

Kamppi, A., Matilainen, L., Maatta, J.-M. & Salminen, E. (2013): Extending IP-XACT to Embedded System HW/SW Integration, System on Chip (SoC).

Kangas, T. (2006): Methods and Implementations for Automated System on Chip Architecture Exploration, PhD Thesis, Tampere University of Technology. Publication 616.

Katayama, T., Saisho, K. & Fukuda, A. (2000): Prototype of the device driver generation system for UNIX-like operating systems, Proceedings. Intern. Symp. on Principles of Software Evolution.

Khan, A.M., Mallet, F., André, C & De Simone, R. (2008): Marte Timing Requirement and Spirit IP-XACT.

King, M., Dave, N. & Arvind (2012): Automatic generation of hardware/software interfaces, SIGPLAN Not., ACM, New York, NY, USA.

King, M., Hicks, J. & Ankcorn, J. (2015): Software-Driven Hardware Development, ACM/SIGDA International Symposium on FPGA.

Königseder, C. & Shea, K. (2014a): Strategies for Topologic and Parametric Rule Application in Automated Design Synthesis using Graph Grammars, American Society of Mechanical Engineers (ASME).

Königseder, C. & Shea, K. (2014b): Systematic rule analysis of generative design grammars, Artificial Intelligence for Engineering Design, Analysis and Manufacturing.

Kruijtzer, W., van der Wolf, P., de Kock, E., Stuyt, J., Ecker, W., Mayer, A., Hustin, S., Amerijckx, C., de Paoli, S. & Vaumorin, E. (2008): Industrial IP Integration Flows based on IP-XACT standards, DATE'08.

Lecomte, S., Guillouard, S., Moy, C., Leray, P. & Soulard, P. (2011): A co-design methodology based on model driven architecture for real time embedded systems, Mathematical and Computer Modelling : Telecommunications Software Engineering: Emerging Methods, Models and Tools.

Lewis, R. (2009): A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing, Computers & Operations Research.

Li, M., Azarm, S. & Aute, V. (2005): A multi-objective genetic algorithm for robust design optimization, GECCO05.

Liu, H., Petracca, M. & Carloni, L. P. (2012): Compositional System-Level Design Exploration with Planning of High-Level Synthesis, DATE.

Lukasiewycz, M., Streubühr, M., Glaß, M., Haubelt, C. & Teich, J. (2009): Combined system synthesis and communication architecture exploration for MPSoCs, DATE.

Lyu, J. (2016): Acceleration of Motion-Detection Application on ZedBoard FPGA using ARM Cortex-A9 CPU and Vision Hardware Accelerators, M.Sc. Thesis, Technical University of Munich.

MARTE (2007): UML Profile for MARTE, Object Management Group.

Nane, R., v. Haastregt, S., Stefanov, T., Kienhuis, B., Sima, V. M. & Bertels, K. (2011): Ip-xact extensions for reconfigurable computing, ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors.

Ochoa-Ruiz, G., Bourennane, E., Rabah, H. & Labbani, O. (2011): High-level Modelling and Automatic Generation of Dynamicaly Reconfigurable Systems, DASIP, Tampere.

OCL ISO/IEC 19507 (2012): http://www.omg.org/spec/OCL/.

OMG - CWM (2003): http://www.omg.org/spec/CWM/.

OMG - MDA (2003): MDA Guide Version 1.0.1, Object Management Group.

OMG - MOF (1997): http://www.omg.org/mof/.

OMG - OCL (2006): http://www.omg.org/spec/OCL/.

OMG - SPT (2005): UML Profile for Schedulability, Performance, and Time, version 1.1, Object Management Group.

OMG - SysML (2008): Systems Modeling Language Specification v1.1, Object Management Group.

OMG - UML4SOC (2005): A UML Profile for SoC, Object Management Group.

Oracle (2014): Java - Computer programming language, Java 8u5.

Paige, R., Kolovos, D., Rose, Louis M., Drivalos, N. & Polack, F. (2009): The design of a conceptual framework and technical infrastructure for model management language engineering, International Conference on Engineering of Complex Computer Systems (ICECCS).

Papyrus (2016): Papyrus Modeling environment 2.0.2 Neon, Eclipse Org.

Paul, J. (2010): FPGA based real-time moving object detection for a mobile platform, M.Sc. Thesis, Technical University of Munich.

Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E. & Heiser, G. (2009): Automatic device driver synthesis with termite, Symposium on Operating Systems Principles, SOSP '09, ACM.

S. Kolovos, D., F. Paige, R. & A. C. Polack, F. (2006): The Epsilon Object Language (EOL), In: Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006, Springer.

SEMATECH (2011): Semiconductor Manufacturing Technology (SEMATECH), http://www.sematech.org.

Stein, F. (2004): Efficient Computation of Optical Flow Using the Census Transform, Joint Pattern Recognition Symposium.

Stephen, N., Thomas, L. & Devin, W. (2015): Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries, Xilinx Application Note XAPP1167 (v3.0).

UML ISO/IEC 19505-1 (2011): http://www.omg.org/spec/UML/.

Vincentelli, A. S., Shukla, S. K., Sztipanovits, J., Yang, G. & Mathaikutty, D. A. (2009): Metamodeling: An Emerging Representation Paradigm for System-Level Design, IEEE Design & Test of Comp.

Wang, S., Malik, S. & Bergamaschi, R. (2003): Modeling and Integration of Peripheral Devices in Embedded Systems, Design, Automation and Test in Europe Conference (DATE).

Xilinx (2012): Xilinx uboot, https://github.com/Xilinx/u-boot-xlnx.

Xilinx (2014): Vivado Design Suite User Guide - High-Level Synthesis, Xilinx Application Note UG902 (v2014.1).

Xilinx (2015): Software-Defined SoC Development Enviroment, http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

Xilinx (2016): Zynq-7000 All Programmable SoC Overview, Xilinx Product Specification DS190 (v1.10).

ZedBoard (2012): http://www.zedboard.org.

List of Corrections