Abstract—UML profiles like SysML and MARTE have been a major research topic in electronic system design, but are mainly applied for specification and analysis in early design phases. High-Level Synthesis (HLS), however, addresses the physical implementation aspect of electronic systems, and thus leads to different requirements on the accuracy of models. For this, modular interfaces are a novel object-oriented synthesizable technique to overcome the conflict between a higher degree of abstraction and necessary details for further synthesis.

In this paper, we present our approach to use SysML as an adequate modeling language for modular interfaces and C/C++/SystemC-based HLS. We extended SysML with annotations for synthesizable SystemC and high-level synthesis constraints and implemented a code generation scheme to achieve design flow automation. Based on the SysML editor Artisan Studio and an industrial case study, we demonstrate the applicability of SysML as a retargetable front-end for HLS design flows.

Index Terms—UML, SysML, SystemC, High-Level Synthesis

I. INTRODUCTION

UML for electronic system design has been studied now for almost a decade [1]. Thus, there exist modeling standards like SysML [2] and MARTE [3] that are adequate for systems modeling and analysis mainly for early design steps. High-level synthesis (HLS) research has its beginnings in the 1980s and achieves now an industrial applicable technology to meet the increasing time-to-market demands for digital systems [4].

This paper introduces our approach to provide a retargetable and automated design flow from SysML to behavioral synthesis (see Figure 1) as one part of our overall HW/SW co-design methodology [5]. We augmented an advanced SysML environment with the support of high-level synthesis constraints to integrate generic C/C++/SystemC-based HLS design flows. We generate synthesizable SystemC code as an intermediate language for simulation as well as for synthesis. Additionally, SystemC testbenches are considered for equivalence checks of the SystemC behavioral input and the synthesized RTL output. We also derive configuration files and makefiles directly from the model allowing one-click design flow automation.

During the evaluation phase it has been shown that for efficient application of SysML targeting HLS requirements, it is essential to reduce the tedious overhead of pin-accurate interface modeling. For this, we developed new extensions for modeling modular interfaces in order to abstract the complex wiring of ports and signals by a synthesizable object-oriented mechanism. We redesigned a complex industrial case study previously implemented in [6], generated the SystemC executable and further synthesized it for FPGA configuration. Our experiments have shown that we reduced the typical interface modeling overhead in SysML of up to 70%.

The remainder of this paper is structured as follows. In the next section we explain our technical contribution in the context of previous work. Section III explains SystemC code generation including the application of modular interfaces. Section IV describes our high-level synthesis extensions offering designers the control over synthesis at modeling time. An overview of our evaluation scenario is presented in section V before section VI ends up with a conclusion.

II. RELATED WORK

Since the introduction of UML2 profiles many initiatives are working on the adaptation of UML for SystemC-based Systems on a Chip (SoC) design. We can identify two main directions: One aims on precise model-to-code transformations for simulation and implementation, and the other one has a rather abstract intent for analysis and allocation.
A comprehensive overview of several model-to-code relationships can be found in [7]. In fact, our framework follows their one-to-one mapping approach for efficiently capturing the semantics of the underlying synthesis subset and allowing round-trip capabilities by common UML tools.

A first investigation on combining UML and SystemC for a one-to-one relationship was done in [8], where they described several benefits especially at early stages of SoC design. Besides further academic work [9][10][11][12] for direct code generation as well as for code generation from the XML Metadata Interchange (XMI) format, there were also industrial efforts. In this context, Fujitsu pushed the development of the OMG UML Profile for SoC [13], and STMicroelectronics introduced their UML for SystemC profile [14] soon after. All of these approaches address the generation of SystemC and map SystemC semantic directly to UML. Although adopting some concepts from these profiles, our approach is more low-level oriented in order to cover high-level synthesis requirements and be applicable for complex systems with a huge amount of wiring. For instance, in contrast to the abstract process modeling in [13] we classify process behavior into sequential and combinatorial logic as described in [15]. However, we avoid the explicit modeling of internal behavior as proposed in [14] because it turned out that writing plain SystemC code at this point is more convenient. Finally, we consider synthesizable SystemC and provide efficient modeling utilities for modular interfaces via metaports as they are not precisely relatable to the standard port/interface concept of UML. In detail metaports are hierarchical ports, which consist again of ports, signals, processes, whereas UML standard ports and provided/required interfaces are based on function call/implementation.

In contrast to the one-to-one mapping approaches, there are OMG standards like MARTE intended to be more general and language independent. For this, MARTE targets rather the functional view of hardware components and does not consider implementation and synthesis details, which we address by our extensions. Nevertheless as the semantic of the extensions is not limited to SysML, so that MARTE could be applied as well, for instance.

Some additional efforts have been spent to combine both directions. For this, in [16][17] they bridge the gap between UML and IP-XACT [18], the de-facto standard for IP integration and management. But as IP-XACT does not explicitly consider high-level synthesis constraints as well, we see our extensions here similarly as a well-suited add-on.

III. SYSTEMC CODE GENERATION AND MODULAR INTERFACES

Our code generation scheme is implemented in Studio’s template-based code generator language and is based on SysML due to its tight integration into the tool. For this, table 1 summarizes our transformation rules from SysML to synthesizable SystemC. The SysML structure can be modeled by three types of SysML diagrams. System engineers should use Block Definition Diagrams (BDD) for system components and their composition. The internal system and communication structure is represented by Internal Block Diagrams (IBD). With SysML Activity Diagrams (AD) one models the SystemC process alignment in terms of sensitivity and port access.

Typical SystemC HLS tools have restrictions on their supported input language constructs. As such synthesizable SystemC modules have to be designed with signal-level ports, which lead to a high modeling overhead if SysML is used in the same way as a Hardware Description Language (HDL). For this, modular interfaces [19] are a novel synthesis technique for hiding recurring pin-accurate interface details from the designer, but still providing them in the model for simulation and synthesis. By means of object-oriented capabilities individual signals and ports are encapsulated into reusable classes and connected with only one binding call. So each modular interface describes a specific communication protocol (e.g. memory access or bus interface) and consists of two complementary sockets, i.e., metaports, as well as an intermediate channel. The communication between a component and his socket is provided by the transaction-level API defined in the socket class. We extend SysML to adapt this mechanism for archiving the following advantages:

- Reducing exceedingly the structural modeling effort in graphical representations of synthesizable SysML models.
- Providing reuse of standard interfaces among multiple design alternatives via model libraries.
- Avoiding redundant development time by strict separation of behavioral and communication sources.

<table>
<thead>
<tr>
<th>SysML Element</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>sc_module/sc_clock</td>
</tr>
<tr>
<td>BlockProperty</td>
<td>Attribute (C++ Instance)/sc_signal/sc_fifo</td>
</tr>
<tr>
<td>StandardPort</td>
<td>sc_port</td>
</tr>
<tr>
<td>FlowPort</td>
<td>sc_in/sc_out/sc_inout</td>
</tr>
<tr>
<td>Connector</td>
<td>Port Binding</td>
</tr>
<tr>
<td>Activity</td>
<td>Static Behavior Structure</td>
</tr>
<tr>
<td>Action Node</td>
<td>Sequential/ combinatorial Process</td>
</tr>
<tr>
<td>Operation</td>
<td>Process Behavior</td>
</tr>
<tr>
<td>Pin/Parameter</td>
<td>Process Sensitivity</td>
</tr>
</tbody>
</table>

Table 1: SystemC code transformation rules

Figure 2: SysML IBD of a modular interface
As shown in Figure 2, we specify synthesizable modular interfaces labeled with <<ModularIO>> by standard SysML objects like blocks, reference properties (specified by dotted lines) and flow ports. As such, the ambient block specifies a channel and the reference properties reflect his associated metaports. From this model our code generator generates a header file composed of three classes (Listing 2). The class DVI_P2P defines a channel including all low-level signals. The two remainder classes, DVI_Initiator and DVI_Target, represent passive metaports, which contain opposed port interfaces, binding functions, and the transaction API. However, a metaport can also be active, i.e., owning multiple SystemC processes and being derived from sc_module.

The application of modular interfaces is done by SysML allocations (Figure 3). For this, a <<Metaport>> stereotype defined in our extensions denotes a SysML standard port as a metaport, which can be allocated from a SysML block for the implementation. A connector between metaports indicate either a high level channel if allocated from a channel implementation or a hierarchical delegation through metaports of the same type. The right allocation of correspondent port/channel types is automatically ensured by means of constraints checking via VBScript in Artisan Studio. The generated SystemC code for the structure of Figure 3 is shown in extracts in Listing 3.
and updates the running script callable from the Artisan Studio.

Table 2: HLS extensions for SysML

<table>
<thead>
<tr>
<th>SysML Element</th>
<th>Stereotype</th>
<th>Tagged Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>&lt;&lt;HLS TopLevel&gt;&gt;</td>
<td>+ compiler</td>
</tr>
<tr>
<td>Block</td>
<td>&lt;&lt;HLS Blackbox&gt;&gt;</td>
<td>+ source</td>
</tr>
<tr>
<td>Block</td>
<td>&lt;&lt;ModularIO&gt;&gt;</td>
<td>+initiator +target</td>
</tr>
<tr>
<td>StandardPort</td>
<td>&lt;&lt;Metaport&gt;&gt;</td>
<td>+type</td>
</tr>
<tr>
<td>FlowPort</td>
<td>&lt;&lt;HLS GlobalReset&gt;&gt;</td>
<td>+ polarity</td>
</tr>
<tr>
<td>BlockProperty</td>
<td>&lt;&lt;HLS Memory&gt;&gt;</td>
<td>+memType +isAsync</td>
</tr>
<tr>
<td>All</td>
<td>&lt;&lt;HLS External&gt;&gt;</td>
<td>+directive +argument</td>
</tr>
</tbody>
</table>

directives, the tools provide us with the opportunity to extend the SysML specification to retarget code generation and control further synthesis. As such, the stereotypes in table 2 address these high-level synthesis directives, which can be grouped into structural and behavioral ones.

For structural constraints as similar to simulation an engineer must be able to specify high-level synthesis entry points. For this, <<HLS TopLevel>> denotes a SysML block as top-level and provides means to define the targeting synthesis compiler. Afterwards subjacent components annotated by <<HLS Blackbox>> are considered to be library modules with only instantiation in the synthesized RTL. For specifying modular interfaces as mentioned before, we define <<ModularIO>> and <<Metaport>> classified as passive or active socket implementation.

In case of HLS behavioral directives, it is required to explicitly specify the reset behavior of internal logic, e.g. registers and outputs. Therefore, <<HLS GlobalReset>> on a SystemC sc_in<bool> port determines reset ports and forces automatically the generation of the SystemC reset logic, i.e., a reset_signal_is and sensitive statement for SC_CTHREAD and SC_METHOD, respectively. Furthermore, memories in abstract functional specifications are typically described in form of C++ arrays. For cycle-accurate timing verification and synthesis, it is important to map these arrays to platform implementation models, e.g., flattened registers or RAM/ROM cells, which are supported by <<HLS Memory>>. Finally, <<HLS External>> on SysML elements provides the capability to generate conditional debug code. Such code is then included in preprocessing statement specified by a directive and an optional argument, e.g. the targeted synthesis subset. Thus, an argument of SC_SYNTHESIS >= 0x123 denotes a synthesis subset version of 1.23.

Our approach currently supports SC Compiler v1.3 as well as SystemCrafter v3.0 without any modifications on the underlying SysML modeling style. For switching between two different compilers, a designer can retarget the code generator by simply redefining the compiler property of <<HLS TopLevel>>. The code generator generates instantaneously tool dependent synthesis constraints, new configuration files and updates the running script callable from the Artisan Studio GUI. The generated SystemC code labels the synthesis directives by a set of C/C++ preprocessing statements. By SC_SYNTHESIS or self-defined arguments, code pieces, that are helpful for debugging and simulation, but which shall not or cannot be synthesized, can be switched off for synthesis as illustrated in listing 3. After code generation, the HLS process can be started by customized context menus.

Listing 3: Conditional SystemC code generation

V. EVALUATION

For evaluation, we applied a complex industrial smart camera system for Automatic License Plate Recognition (ALPR), which is an image-processing technology used to identify vehicles by their license plates. The original platform implementation without modular interfaces was undertaken by Thales Security Solutions and Services in a joint European project [6] and was completely modeled in Artisan Studio using our HW/SW co-design methodology from [5]. After SystemC code generation, synthesis to RTL was done by the Agility Compiler. Furthermore, they generate the bitstream with Xilinx ISE toolchain and upload it to a Virtex-5 FPGA integrated into their ML507 Development Platform for prototyping.

The smart camera system is built in multiple pipeline stages of combined hardware and software components. The abstract functionality can be roughly divided into the three following phases:

- Image Acquisition & Camera Control (HW & SW)
- Image Processing & Labeling (HW & SW)
- Optical Character Recognition (OCR) Functionality (SW)

The first phase is in charge of capturing the pixel stream and setting up the camera by means of exposure time and gain computation calculated on the grabbed images. During the image processing and labeling phase several filters in form of mathematical morphological operations are passed through, which performs the base algorithms of license plate detection. These filters are applied to detect regions containing a high density of small elements, which is a well-suited characteristic of text like on license plates. The top-level structure of the developed image processor is shown in Figure 4. It is the main core of this phase and can be launched in the following different variants:

- Basic dilation/erosion operators with handling of variable window size.
- Composition of dilation and erosion for opening and closing filters
- Top Hat algorithms based on opening and closing with subtraction of the original image
Afterwards, the identified regions are passed to the OCR application running on a PowerPC440 to read the actual plate number or discard non text.

For demonstrating the benefits of modular interfaces modeling and retargetable code generation, we focused on the morphological filter communication infrastructure as one part of the smart camera system and redesigned the entire image processor. However, HW/SW interfaces, in our case the IBM CoreConnect architecture, are also an interesting area of application for modular interfaces. An overview of the communication structure is shown in Table 3.

<table>
<thead>
<tr>
<th></th>
<th>SysML Blocks</th>
<th>SysML Ports</th>
<th>SysML Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Model</td>
<td>20</td>
<td>292</td>
<td>166</td>
</tr>
<tr>
<td>Model using Modular Interfaces (without Reuse)</td>
<td>44</td>
<td>157 (46%)</td>
<td>99 (40%)</td>
</tr>
<tr>
<td>Model using Modular Interfaces (with Reuse)</td>
<td>20</td>
<td>69 (76%)</td>
<td>55 (66%)</td>
</tr>
<tr>
<td>Modular Interfaces</td>
<td>24(8x3)</td>
<td>88</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 3: Communication structure reduction of the ALPR image processor design

The original design from Thales is composed of 20 SysML blocks whereof 15 are taken into account for synthesis and 5 for verification purposes. For modeling the communication structure they required a total of 292 SysML flowports and modeled 166 port bindings in form of SysML connectors. With our new extensions we create 8 additional modular interfaces as summarized in the last row and integrate them into the model. Afterwards, the image processor diagram corresponds to Figure 5, which fits much more in the usual UML/SysML modeling appreciation, i.e., analysis and documentation friendly. Taking the implementation of modular interface as new effort (without Reuse), we reached a reduction of 46%. If modular interfaces considered as coming from a preexisting model library, the reduction increases up to 76%. Thus using our extensions engineers could expect an overhead modeling reduction close to 70% after their first design, which is an efficient improvement in designing synthesizable SysML models.

VI. CONCLUSION

In this paper, we introduced our retargetable and automated hardware design flow from SysML modeling to SystemC-based HLS. We enhanced Artisan Studios SysML modeling environment so that engineers could efficiently specify and control hardware synthesis via modeled high-level directives. Due to the fact that SysML comes with the default configuration of Artisan Studio, we defined our stereotypes in an additional profile package on top of SysML, which can be simply loaded into the tool. Based on dedicated transformation rules, we implemented a retargetable code generator for synthesizable SystemC as well as design flow automation. The code generation itself executes by permanently observing modifications of the model and updates the code on demand. Via configuration files and preprocessing directives the generated SystemC code is efficiently retargetable for multiple HLS compilers. For evaluation we partly redesigned an industrial smart camera system, which passed the whole design flow from SysML modeling, over SystemC code generation to synthesis. Our measurements show a reduction of the communication structure modeling overhead by almost 2/3.
REFERENCES