Building Code Generators for Reuse - Demonstrated by a SystemC Generator

Saad Siddiqui, Intel Deutschland, Neubiberg, Germany (*saad.siddiqui@intel.com*)

Ulrich Nageldinger, , Intel Deutschland, Neubiberg, Germany (*ulrich.nageldinger@intel.com*)

***Abstract*—This paper presents actual experience in the conversion and development of a SystemC code generator alongside the project using it. The baseline was an existing generator for a different code generation framework using a different data model. Thus the chosen approach to create a working solution in a relatively short time had also to reuse as much as possible of existing generator code. Our solution to address these requirements uses a layered setup of data preparation and code generation functionality, which turned out to be well suited for fast generator creation and efficient maintenance. This approach can be considered general beyond the initial scope of the SystemC target.**

Keywords*— *Code generation, Virtual Prototype, SystemC

I. Introduction

Virtual Prototypes (executable models of hardware) are a successful way to improve a classic SoC development process as they allow for a shift-left of the Software development before the RTL is available (in contrast to FPGA-based solutions). In this process, the Virtual Prototype appears as one further deliverable in parallel to RTL and Software. A comparison between a sequential SoC development process and a process using a VP is shown in Figure 1.

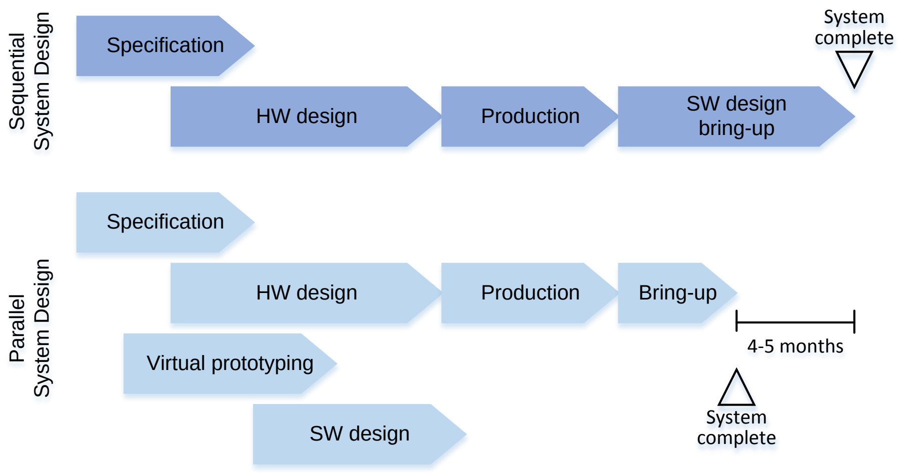

Figure 1. Sequential and parallel SoC development process

The crucial part of this process is a mandatory consistency of the HW/SW interface between the involved collaterals. This means that the description (names, types, addresses,...) of the Control Registers must match between the VP model, the final Hardware (represented by RTL collateral) and the Software (typically represented by access macros). So in order to achieve this shift left, an efficient development methodology is needed which is tightly coupled to the overall SoC process. Typically, a machine-readable form of the register descriptions in combination with automatic code generation is employed.

Such a flow puts up the additional need for a powerful *code generation methodology*. Whenever new requirements to the actual design occur in the specification phase (like a new type of register, a new side effect when a field is written, new requirements regarding width of registers, etc.) the code generation for all collaterals must be adapted in a reasonable time. So an approach is needed for efficient generator development.

The work described in this paper originated from the following scenario: For a running project we needed to provide a generation flow using register descriptions in SystemRDL [1] and requiring SystemC [2] collaterals compliant to an internal SystemC register library. A flow to do this was available using an existing XML-based solution as an intermediate format (together with an according SystemC generator), i.e., the flow included conversion from SystemRDL to this format and then the actual SystemC generation. However, it turned out that some customer requirements regarding specific elements (in this case: specific types of registers and bus interfaces) could not be met by the data model of the intermediate format. However, a new intermediate format and a path from SystemRDL to this new format was available, but no generator for SystemC.

This paper describes our approach to provide a native SystemC generator and at the same time create a reusable basis for future generators. In order to get fast results we had also to reuse as much as possible from the old generator. This paper discusses our experiences with this work.

In the following, we will motivate a general Single-Source Flow as a basis for our approach. We will then describe our generator architecture. The paper is concluded by results achieved and an outlook to future work.

II. Single-Source Flow

A major issue to be addressed for VP development is the need to ensure consistency of the VP model with the actual hardware. While the functional equivalence has to be ensured by e.g., Co-Verification, the consistency of specification information (like register names and offsets, base addresses, interface names and types, etc.) can be ensured during development by a Single-Source Flow as shown in below Figure 2.

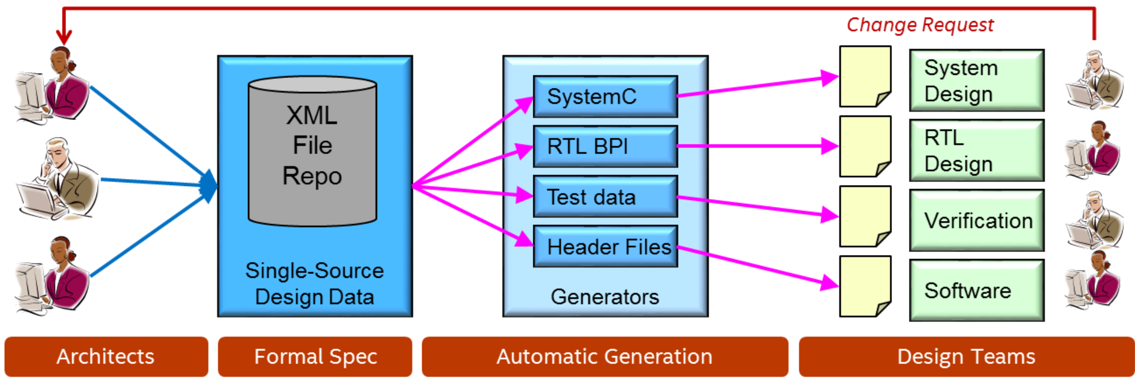

Figure 2. Single-Source flow

The basic concept of a Single-Source flow is to capture the specification information in a formalized way (XML files) and create according collateral for the design tasks through automated code generation.

It should be noted that a Single-Source Flow does not only contain a code generation tooling but also an efficient handling and release process for the formalized XML files. It must be ensured that:

- All necessary data is captured by architects and the data is correct.
- There is a well-known place for the formalized specs and all project members take it from that place. A well-defined release process for the formalized specs must be established, including a build system which automatically propagates changes to collaterals throughout the project.
- While local copies are allowed for debugging, any persistent changes to the formalized spec must be integrated into the Single-Source XML files, and the Single-Source files are the only ones to be propagated to the whole project.

It can be seen, that such a process needs significant discipline and more effort at the specification side (architects), but the advantages justify this effort, especially if a VP with an essential demand for consistency to the hardware is developed:

- Consistency between hardware, VP, software, verification and documentation. This includes reducing unpleasant bugs found typically towards the end of the project. Possible problems without a Single-Source process include teams using different versions of the specification, or even that the written specification differs from the hardware.
- Fast propagation of spec changes (especially: fast propagation of bug fixes in the specification)

III. Flexible generator development shown by SystemC Generator

Once a formalized specification is available, it can be used to generate a wide variety of collaterals. Regarding these generators it has turned out that there is the need for both "standardized" generators (e.g., documentation, RTL) as well as for user-defined generators. While the first type of generators is typically handled by central support teams (who also provide the underlying methodology) the second type requires a convenient way for end-users to write their own generators.

To fulfill this requirement, one approach having proven to be quite successful in the original framework of the SystemC generator is the combination of a high-level API to retrieve the needed information and a template engine supporting a scripting language (an approach known from content management systems). The use of a scripting language is already a major convenience for both end-users and central generator writers. A template engine adds a further productivity boost as it allows to represent the target collateral in a markup-style way, i.e., fixed text does not change and special constructs allow for retrieval of data elements, traversing over lists, conditional text or embedding of scripts for more sophisticated calculations.

The approach shown here thus uses a binding of the API of the target framework to the Python scripting language, a high-level API in Python to prepare the data and an open-source template engine (which is also written in Python).

This approach is exemplified by the SystemC generator shown in Figure 3:

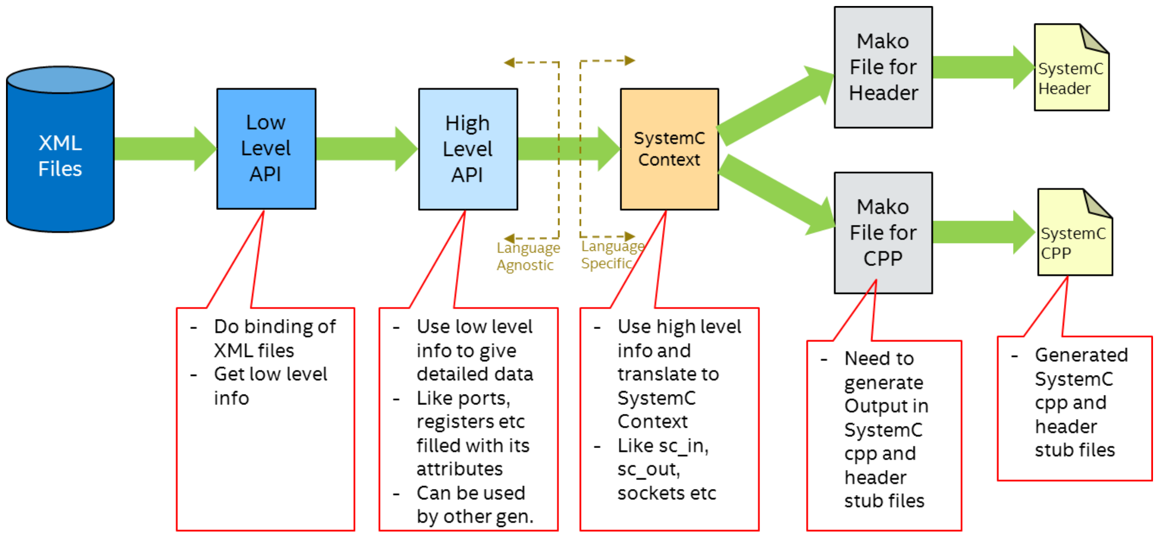
XML Files
Low Level API
High Level API
SystemC Context
Mako File for Header
Mako File for CPP
SystemC Header
SystemC CPP
Language Agnostic
Language Specific
- Do binding of XML files
- Get low level info
- Use low level info to give detailed data
- Like ports, registers etc filled with its attributes
- Can be used by other gen.
- Use high level info and translate to SystemC Context
- Like sc_in, sc_out, sockets etc
- Need to generate Output in SystemC cpp and header stub files
- Generated SystemC cpp and header stub files

Figure 3. Architecture of SystemC Generator

A. *Low-Level-API*

The low-level API provides only the basic functions for handling of the XML files containing the formalized description like opening of these files, setting up the environment and getting/setting arbitrary properties stored in the files. This API level is oblivious of a specific data model. It provides the same interfacing functionality to e.g. register descriptions and pin lists.

It should be noted that both XML formats involved in the task described are proprietary formats to describe design data in a formal way. We are aware that standard solutions like IP-XACT [3] describe similar data. However, while IP-XACT targets IP-packaging and focuses on information needed to integrate existing IPs, the proprietary formats target (in-house) code generation for IPs. Thus these formats carry more information and feature advanced mechanisms to foster the reuse of these formalized descriptions as a part of the IP specification.

B. High-Level-API:

Based on the low-level functionality, the next layer contains general higher-level functions for a specific "topic" or type of data which is in this case register descriptions as well as port information. Functions typically include traversal of the description tree and calculating common register attributes (e.g., the effective address) as well as handling of sequencing information, preparing hierarchical names and others. An important property of this level is its agnosticism of any target collateral.

In the following some example high-level-api functions are described. We will illustrate the functions by their relation to SystemC although this API level is target-independent:

- Return a list of "physical" registers of a module: As one specific register of a module may be visible on one or more SystemC sockets, a list of the actually existing registers (independent of the socket visibility) is needed to create the instances.
- Return a list of registers visible on a certain socket: When it comes to actual registration of registers at a SystemC socket, a second view on the set of registers is needed showing the subset visible on a specified socket. It should be noted, that this function is also needed for collaterals like Software header files (in contrast to the above function retutning the list of all physical registers) as Software can only access registers through a bus interface.
- Return a list of bit fields in a register: Depending on the collateral, reserved bitfields or bit field gaps may or may not be filtered out automatically. For SystemC typically both reserved fields and gaps are removed.
- Return a list of sockets and ports: While a bus interface (which may comprise lots of RT-level ports) is modeled in Transaction-Level SystemC as one socket element, individual ports (like clock, reset, IP-specific ports like TxD, RxD for a serial interface,...) need to be handled separately so the API has to take care of handling both types of perimeter connections.

C. SystemC Context

While simple generators may directly run on the High-level API, more sophisticated ones may also need collateral-specific processing. For the SystemC generator this additional information is prepared by a layer called SystemC Context. We use Aspect Oriented Programming to add language-specific information to the existing elements received from the High-Level API. In this sense, the SystemC Context does a Model-to-Model transformation from the original data model prepared by the High-Level API to a language-specific model which contains all information needed for the output collateral. Examples for such information are described in the following:

- Element naming: For the generation of SystemC elements (for registers, ports, .sockets,...) a specific naming convention has to be followed and needed datatypes have to be calculated. Names may also depend on the type of element and its position in the hierarchy.
- Class information: The element properties have to be mapped to the correct class of the underlying SystemC library. As this information is needed several times, it is prepared once and stored at the element.

D. Mako Files

After all needed information is available, the actual format of the output files (in this case C++ source and header files) needs to be manifested. It has shown that a template engine (e.g., the Mako engine [4]) known from application in Content Management Systems gives significant benefits in describing the output format over a pure-code implementation with explicit print statements.

The Mako files do not perform large calculation tasks (although it is possible through embedded Python code) but contain templates for the generation of code parts. The template engine approach allows for structuring the generator according to the output file instead of the data model which makes development and maintenance more convenient than pure scripting code.

IV. RESULTS

The previous flow from SystemRDL to SystemC was using a proprietary framework which inherited limitations based on its data model. This lead to the need for workarounds and loss of information during the (multiple) conversion flows.

In comparison, the target framework for our approach is by itself better suited for this flow due to the following reasons:

- The conversion path from SystemRDL is much more streamlined than to the old framework, so that we had a clearer data basis to work on without inherent workarounds for unsupported features.
- The data model of the target framework is a superset of the old framework, so we did not have additional migration problems caused by needed new workarounds. In this sense, the actual migration effort could focus on code rewrite and optimizing the reuse of existing code.

The selected approach therefore consisted of these steps:

- First get a working version of the new generator, reusing as much from the old one as possible. This version would of course have the restrictions of the old generator.
- Based on the first version we would enhance the generator step by step with requirements from the running project.

The first step could be done by rewriting and adapting existing Python/template code for the High-Level-API, the SystemC Context and the template files in a reasonably short time of four weeks (the low-level Python binding was already available). Most work went as expected into the High-Level-API while the SC-Context and the template files needed mostly only refactoring of names. Being able to mostly reuse the template files proved especially useful as we could get compilable output files in an early stage.

In the second phase, the extension of the base generator, the actual focus of development effort into the different layers depended on the actual requirement and is more heterogeneous. However, the chosen layering of the generator functionality proved efficient and reasonable, as we could clearly put new functionality to a specific layer without needing to touch any other one, thus keeping the development effort for each new feature low. As we could release some feature enhancements within few days we consider our approach very well suited for maintainability of generators.

We are aware that the real value of our approach needs to be proven in the future for the migration of more generators from the old framework to the new one. Our primary objective was to create a reusable baseline for generator development and the actual reuse factor will be verified not before more generator migrations are done.

A general drawback of our approach - caused by the high reuse of existing Python code - is the long runtime of generation (~ 15 min. for a complex system with 9.3k registers). However, this is mainly caused by a "straight-forward", non-optimized data access to the Low-level API. We expected this as we bought the quick availability of a working solution with a suboptimal runtime. This drawback can be addressed by optimizing the code towards the new API as well as implementing according caching techniques.

V. Future Work

The SystemC generator described in this paper will still being enhanced. So we expect a further solidification of the lower layers, making them more robust for reuse. In parallel, a streamlining of the code (High-level API up to Mako files) will take place in order to improve the runtime of the generation.

For the long-term, the conversion of further generators from the old framework using this methodology is planned based on the work described here, which will enable use of the more advanced solution and also the replacement of the obsoleted framework.

VI. Summary

We presented an approach to generally implement code generators building on a Low-level API, using multiple layers of functionality. In our approach we tried to focus on these requirements: Allowing for migration of existing code, getting quickly to a working solution, building a reuseable infrastructure for future generators and ease the generator maintenance. We found that our approach does satisfy these requirements due to the use of Python for the main functionality and a template engine to describe the collateral format. Future work includes hardening of the lower levels in order to reduce runtime as well as verifying the concept for future generators.

References

[1] http://www.accellera.org/activities/working-groups/systemrdl

[2] http://www.accellera.org/activities/working-groups/systemc-language

[3] http://www.accellera.org/activities/working-groups/ip-xact

[4] http://www.makotemplates.org