Abstract

We present a reuse scenario for the VHDL-based hardware design flow based on a library of extremely flexible parameterizable components supplemented by the support of most of the phases of the design process ranging from specification refinement and modeling, over simulation and synthesis, down to gate-level verification and HW test support.

This reuse scenario has been inspired by the abstraction and modeling capabilities of VHDL and can be best characterized as know-how reuse.

Currently, our scenario provides reuse for components with a complexity ranging from 0.1K to 15K gates. Qualitative and quantitative results demonstrate the potential and the feasibility of our reuse scenario.

1 Introduction

Keeping pace with the market and the increasing complexity of ASICs and HW systems means to continuously improve the design processes. To achieve the required increase in engineering productivity and design quality, VHDL and synthesis were added to the design process in the past. Today, reuse concepts are among the most promising techniques to be introduced for this purpose [GiCa93, Runn94, Dutt94].

The idea of reuse is not new. HW designers have been re-using (portions of) their design data such as layouts, netlists, or HDL code without significant modifications whenever this has been possible. Design reuse, however, fails if the design portion is not flexible enough, if it is not well-documented, if it saves effort marginal in comparison to the effort needed for the standard design technique, and if there is no well-defined reuse process.

The reuse scenario presented in this paper has been enabled by the programming capabilities of VHDL. VHDL allows to cover a wide range of applications with a single flexible model. Furthermore many design expertises of the design flow can be integrated into the VHDL model. Our reuse scenario is therefore best characterized as know-how reuse.

The scenario relies on a library of reuse components (see, Figure 1) i.e. reusable (extremely flexible, parameterizable and self-generating) components [Prei94, PrRo95] each combined with a tailored reuse process. All reuse components are designed for reuse, and cover a considerably large ASIC portion each.

The reuse process is based on a HyperText documentation and checklists, guiding the designer through all design phases of the application of the reuse component. This includes for each component a specification refinement strategy, modeling guidelines, a verification strategy incl. test concepts, a flexible unit testbench with standard test cases and gate-level verification support, as well as tailored synthesis scripts.

1.1 Related Work

The term design reuse has been used for several concepts. Structured design with building blocks is one view on design reuse. Accordingly, Keutzer [Keut89] and Pawlak et al. [BILO93] proposed module or component generators, respectively, to enable the (re-)use of complex building blocks. A more SW-oriented view focuses on the reusability of VHDL code. A first step towards reusable code are coding style guidelines and tools to support these [HaMa94, PKSS93]. Girczyc et al. [GiCa93] identify four basic categories of design reuse: design exchange without changes, design evolution involving changes as well as technology-independence (cf. [FiDH93]), design knowledge encapsulation for intellectual property protection under broad distribution, and reprogrammability of FPGAs.

Component reuse has a great tradition for microprocessor based systems including RAMs, ALUs, I/Os and datapath components (e.g. adder, multiplier). The main emphasis hereby is the definition of component functionalities and their efficient structure.

Runner [Runn94] presents a scenario, similar to ours in that all aspects of data that a designer refers to are subject to reuse. His scenario, however, includes VHDL model parameters for scalability only (e.g., bit width, number of instances) to satisfy the needs of a synthesis-dominated use of the component library [DW94]. In contrast to that, our reuse scenario refers to the complete VHDL-based design flow, targets a higher degree of parameterizability (e.g., functional variants, self-generation) and extends towards know-how reuse.

1.2 Overview

The rest of the paper is organized as follows. Section 2 starts with an elaboration on the application domain and requirements that triggered our reuse scenario. The details and implementation aspects are described in Section 3. Section 4 presents qualitative and quantitative results from the application of our approach in a large Siemens design center, and Section 5 concludes with pros and cons of the approach.
2 Requirements for Reuse

2.1 Application Domain

A useful taxonomy for ASIC blocks is control-dominated versus datapath-oriented. Control-dominated blocks consist of standard components (i.e., complex components with a high degree of markedness) and control blocks (FSMs) controlling the activity of these. While architectural synthesis is the appropriate technique for handling datapath-oriented blocks and FSM-based tools are appropriate for control blocks, reuse turned out to be the best technique for handling standard components.

Another reason for reusing standard components can be found in how ASIC complexity evolves. Since more and more functionality will be concentrated on a single chip, the complexity growth of control-dominated blocks results in an increasing number of the standard components rather than in an increase of the standard components functional complexity.

2.2 Hardware Design Process

The goal of design reuse is the increase of engineering productivity, i.e., the decrease of development time and the increase of design quality, next time a similar component has to be designed. Hence, reusing a component must be cheaper than designing it from scratch, and the overhead in developing a reusable component must pay off within a reasonable number of re-uses.

First of all, design reuse can only be established if the components are designed for reuse. Passing the design from one tool or design step to the next is crucial. Only with reuse and the overall design flow in mind, a component can be modeled incorporating all the functionality needed for the component’s different applications.

Second, the goals of design reuse can only be reached, if the process of reusing the component is considered as well, i.e., if a designer can successfully integrate the reusable component into the relevant design process. To ease the use of reusable components, the following requirements have to be satisfied:

- **Immediate identification**: The designer must be able to immediately decide whether the required component is available, and if so, whether it provides all the functionality needed and if it meets all the constraints.
- **Direct access**: The designer must have access to the reuse component. The application of the reuse component must be documented and supported.
- **Verification support**: The reuse component must be pre-verified and the designer must be able to run her/his own functional tests.

The benefits of covering all these aspects are:

- a well-defined procedure for refining the specification,
- simplification in getting into the modeling theory of the component,
- reduction of the modeling process to parameter specification,
- shortening of the functional testing phase due to pre-verified reusable components and unit test environments,
- synthesis process automated by scripts,
- access to expert’s knowledge to avoid traps and design iterations,
- know-how driven guidance through the design process.

3 The Reuse Scenario

Figure 1 illustrates our reuse scenario for one reusable component which has been drawn from the requirements listed in Section 2.2.

![Figure 1: Schematic of a reuse component.](image)

The reusability of the components is established by three major elements: the model, the verification support, and the documentation. All of them are integrated in a Hypertext based reuse environment. If one of the three elements is missing or the environment does not correctly control their interactions, reuse will fail. This scenario will be described in more detail in the following section.

3.1 The Model

The heart of each reuse component is the VHDL model which covers a wide range of applications from a single source and which controls the relevant tools through pragmas. In the following, we explain the VHDL modeling concepts underlying our approach.

3.1.1 Requirements for Parameterization

The quality of the reusable model (component) lies in an extended parameterization of the component’s functionality. Parameterization may range from simple to very complex aspects. These include the definition of the:

- **Logic functionality**: i.e., the specification of functionality related to the logic level (e.g., specification of active signal levels).
- **Component structure**: this includes insertion, selection, and arrangement of sub-components, e.g., register files, data width, pipeline register files, BISTs.
• **Component functionality**: specification of specifics related to the component functionality, e.g. register and RESET type, signal coding, polynomials, degree of parallelism, definition of architectural representations.

Moreover, non-functional aspects of the reusable component must be supported by the parameter set.

• **Design management information**: e.g. model type, reports, revision control, automated checks.

### 3.1.2 VHDL Constructs for Reusability

VHDL was chosen since it provides powerful parameterization capabilities, and on top of these the so-called self-generation feature which allows the modeling of extremely flexible functionality. See [Prei94, PrRo95, JoVT91] for details.

### 3.1.3 Modeling Styles

The basic modeling style for a reuse component is the *fully parameterizable* model, i.e., all the functionality needed can be completely controlled by parameters only.

For very complex components the number of different applications and therefore the number of parameters can exceed a manageable size. Moreover, the management of an undefined number of ports creates problems w.r.t. tools and code readability. In this case, a second modeling style is preferred: a *template-based* model. The template is a construction kit to build the model from fully parameterizable reuse components. For very complex components, the template-based model provides a good balance between parameterizability and parameter manageability.

### 3.1.4 Modeling of Supplementary Aspects

Features extending the functionalities of the reusable components were managed as follows:

- With reuse components with up to 30 parameters each it is easy to "get lost in the parameter space". Therefore, report files are generated containing information of the component-instance functionality.

- Built-in support of revision control for development and maintenance of the reusable components is provided. Both aspects have been designed using the TEXTIO mechanism of VHDL.

- Furthermore, the compliance to our coding style and modeling guidelines [PKSS93] aid the application of the reuse components.

### 3.1.5 Implementation Aspects

To demonstrate the effectiveness of our reuse scenario, we modeled several components. Examples are parallel CRCs, CRC evaluation, complex register files with access conflict resolution, μP-Interface. The μP-Interface has been designed as a template-based model.

A detailed analysis of several ASIC designs and interviews with designers allowed to determine the component selection and their functional specification.

Unfortunately, not all tools in the design process do support all the required VHDL constructs. For a successful implementation of the reuse scenario additional VHDL modeling techniques (e.g., conversion functions, abstract coding) and work-arounds had to be implemented.

### 3.2 The Verification

It would be nice, if the designer could reuse a completely tested and error-free reuse component. Already the complete test of a non-parameterizable component fails due to the complexity of the test cases. Parameterizable components increase the number of the test cases with each additional parameter [PrRo95]. Moreover the template-based approach makes it impossible to cover all possible designer modifications.

Checklists guide the designer through the different design phases. But they are only a method to prevent composition errors rather than a method of correctness-by-construction. The designer can expect a pre-verified, but never completely tested, reuse component to be released in the library. The final verification of the component, whether fully parameterized or assembled in the template approach, must be left to the re-user. As an aid to that step the reuse component contains a unit testbench (for simulation, which is the most important verification step up to now) together with a checklist which guides the designer through the verification process.

#### 3.2.1 Functional Test in the Development Phase

![Figure 2: Global testbench structure with parameters as deferred constants and automated verification.](image)

Our functional test concept for the development phase is as follows:

- **Strict separation of the development and the test tasks**: Both teams are separated. The test team knows the external behavior of the component only, but nothing about implementation details or the structure of the component to ensure that implementation-independent test cases are chosen.

- **Bottom-up test approach** (borrowed from SW test methods) [Beiz90, Ligg90]: Small base units are tested as completely as possible first (unit test), before the integration of these units into larger modules (unit integration test) and these modules into components.
(component test) will be tested. This forces the creation of testbenches for each single base unit.

Testbenches must be highly comfortable concerning the adaption of the parameter set, the stimulation and the validation techniques.

The main structure of our testbenches is shown in Figure 2. To allow for a quick and easy adaptation of the component’s parameters in the testbench, parameters are declared as deferred constants in a package body. This simplifies the simulation and validation of orthogonal parameter settings with almost automatic adaption of the stimulation and validation code. Every testbench contains:

- **stimulation** techniques, which can be an algorithmic, i.e. macro-oriented, or a table-oriented method, respectively. The testbench contains a set of stimuli.
- **validation** techniques, which include timing and functional checks. A set of expected values is already prepared.
- **protocol** generation, which logs the stimulation and informs the designer about results of functional and timing checks.

### 3.2.2 Functional Test in the Reuse Phase

The final test of the fully parameterized or assembled template-based components, respectively, is part of the reuse component. A structured test approach requires a unit test before the integration of the unit into her/his system.

Often unit test are avoided due to the required effort in creating additional testbenches. To enable unit test and therefore to increase the design process reliability, a highly comfortable and easy-to-use testbench is prepared for each reuse component. Fortunately, the testbench needed for the unit test already exists from the development phase. Hence, component reuse also includes testbench reuse.

In the case of a fully parameterizable reuse component the testbench is also fully parameterizable. Stimulation and validation are automatically adapted by the parameters of the reuse component. There is no need for manual adaption of the testbench. Basic test cases are also prepared. Of course, user-defined test cases can be added.

In the case of a reuse component using the template-based approach the already existing unit integration testbench from the development phase is prepared as a template. As for the model itself, a checklist guides the user through the adaption of the template-based testbench and the composition of the testbench for her/his composed model.

### 3.2.3 Support of HW Test

First of all, BIST structures have been implemented as part of the reusable components where it makes sense. Of course, these BIST structures can be controlled and built in by parameters.

Second, the models are prepared to support scan path strategies, i.e. the generation of scan path.

Finally, parts of the verification environment for the functional test can be re-used for the generation of HW test vectors.

### 3.3 The Documentation

The common meaning of the term documentation is just to comment the source code and to describe implementation specifics of a component already designed.

Here, the term documentation means a “user manual” for applying the reuse component. The purpose of the documentation is to guide the designer through the complete process of applying a reuse component. This includes:

- **Support information** e.g. history, environment, revision control.
- **Component description** e.g. specification, theory, structure, functionality, parameterization, implementation specifics.
- **Modeling information and guidelines** e.g. use of instantiations and templates, parameterization, refined specifications.
- **Verification information and guidelines** e.g. testbench concepts, stimulation, validation.
- **Synthesis guidelines** e.g. use of methodologies and scripts
- **Examples.
- **Technical data.
- **Reference pages** for signals and parameters.

The complete documentation is available as a HyperText document. Creating such a documentation incl. reuse guidelines, takes a major part of the development time of the reusable components (see Table 3 and Table 4).

### 3.4 The Environment

The Environment is not a tool but a modus operandi and a strictly committed process w.r.t. the following aspects:

- definition of supported tools, e.g., for simulation, synthesis, documentation, data management, revision control, and their usage.
- definition of the documentation and data sheet standards.
- definition of operational conditions, e.g. design platforms, information protection, authorization for usage.
- definition of user support, e.g., on-line help documentation, revision control, information flow.
- definition of modeling rules and coding style for the VHDL models. These guarantee the support of the complete design flow and all necessary tools, improve the readability and ease the modifiability.

### 4 Results

The library of reuse components currently contains 18 building blocks ranging from 160 to 2700 Lines of VHDL code (LoC) enabling the reuse of HW components at complexities ranging from 100 up to 15,000 gate equivalents.

Examples are parallel CRCs, CRC evaluation, register files with access conflict resolution, and a µP-Interface.

The µP-Interface is a template-based component whose complexity depends on the number of sub-components assembled. All other components are fully parameterizable and are partly used as sub-components in the more complex models.

The user is guided through the design process from specification refinement, modeling, verification, synthesis, down to HW test generation by checklists with up to 40 items each, and with 200 items for the template-based component.

Figure 1 shows the structure of each of the reuse components including all the required aspects. The heart is the
VHDL model integrated into the design flow, i.e. the refined specification, templates for modeling, testbenches and synthesis. All design steps and functionality aspects are covered by checklists and HyperText documentations. The components have been successfully used in several ASIC projects.

4.1 Portability
The components have been ported to several different VHDL simulators [Tools]. Porting to simulators, supporting full VHDL, did not create any problem.

4.2 Analysis of the Parameter Sets
To compare the degree of parameterization provided in our approach with the degree provided by a commercially available method [DW94], we examined the types of the parameters provided. We distinguish the following categories: component structure (width), logic functionality (level), component functionality (function) (refer to Section 3.1.1).

The average distribution of parameters over these categories is shown in Figure 3 together with the number of ports, the number of lines of (VHDL-) code and the number of gates for a small implementation.

<table>
<thead>
<tr>
<th># parameters in category:</th>
<th># ports</th>
<th>LoC</th>
<th># gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>9.71</td>
<td>510</td>
<td>250</td>
</tr>
<tr>
<td>level</td>
<td>6.65</td>
<td>1</td>
<td>(6%)</td>
</tr>
<tr>
<td>function</td>
<td>8.47</td>
<td>9</td>
<td>(52%)</td>
</tr>
<tr>
<td>param.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>template</td>
<td>2</td>
<td>13</td>
<td>2700</td>
</tr>
<tr>
<td></td>
<td>(11%)</td>
<td>9</td>
<td>(47%)</td>
</tr>
<tr>
<td></td>
<td>(42%)</td>
<td>8</td>
<td>(42%)</td>
</tr>
</tbody>
</table>

Figure 3: Average distribution of parameters per component over different parameter categories.

i. Small realistic implementation of the components.
ii. Average values of 17 fully parameterizable components.
iii. Template-based component.

The conventional methods provide (almost) only scalability parameters that adjust the bitwidth of signals and structures. Our results show that the number of scalability parameters (bitwidth parameters) is very low (6% - 11%) in comparison to the overall parameter space needed for reuse components.

The functional parameters add new features to the components and cause drastic variations in the number of gates.

The number of parameters weakly depends on the number of ports, as the number of LoC VHDL also weakly depends on the number of parameters. In some situations, the number of LoC VHDL is unnecessarily large. This is due to synthesis tool restrictions and the corresponding VHDL-based work-arounds.

The template based method (see Figure 3) is a realistic but very small implementation of the µP-interface. Practical implementations of this component may reach the complexity of 15 000 gates, which of course also leads to an increased number of ports and parameters.

4.2.1 Testbench Results
Figure 4 shows a selection of the testbench results. In case of unit A and B, only the simulation time for the unit test must be spent by the designer, since the testbench is automatically adapted by the component’s parameter set. Furthermore unit B the testbench includes 1100 prepared test cases. The 2700 LoC template-based model is released with a 4000 LoC testbench template containing 900 prepared test cases. The adoption of the testbench can be done in about one day, which is a very small fraction of the effort needed for creating the testbench from scratch.

<table>
<thead>
<tr>
<th>unit</th>
<th># LoC of reuse testbench</th>
<th># prepared test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>algorithm</td>
</tr>
<tr>
<td>B</td>
<td>2700</td>
<td>1100</td>
</tr>
<tr>
<td>C</td>
<td>4000</td>
<td>900</td>
</tr>
</tbody>
</table>

Figure 4: Selection of testbench results.

i. A small realistic implementation.

4.3 Effectivity of the Reuse Scenario
Figure 5 and Figure 6 compare the effort for developing a single instance from scratch, the effort for developing a reuse component, and the effort for using a reuse component.

<table>
<thead>
<tr>
<th>time [%]</th>
<th>design of a single instance</th>
<th>design of a reuse component</th>
<th>use of a reuse component</th>
</tr>
</thead>
<tbody>
<tr>
<td>preparation</td>
<td>13</td>
<td>65</td>
<td>3</td>
</tr>
<tr>
<td>modeling</td>
<td>45</td>
<td>125</td>
<td>3</td>
</tr>
<tr>
<td>verification</td>
<td>38</td>
<td>125</td>
<td>3</td>
</tr>
<tr>
<td>documentation</td>
<td>4 (1%)</td>
<td>125 (11%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>sum</td>
<td>100</td>
<td>440</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 5: Standardized time effort for developing an instance from scratch, developing a fully parameterizable reuse component, and using this reuse component.

<table>
<thead>
<tr>
<th>time [%]</th>
<th>design of a single instance</th>
<th>design of a reuse component</th>
<th>use of a reuse component</th>
</tr>
</thead>
<tbody>
<tr>
<td>preparation</td>
<td>13</td>
<td>26</td>
<td>7</td>
</tr>
<tr>
<td>modeling</td>
<td>40</td>
<td>67</td>
<td>13</td>
</tr>
<tr>
<td>verification</td>
<td>40</td>
<td>80</td>
<td>13</td>
</tr>
<tr>
<td>documentation</td>
<td>7 (1%)</td>
<td>52 (2%)</td>
<td>2 (1%)</td>
</tr>
<tr>
<td>sum</td>
<td>100</td>
<td>225</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure 6: Standardized time effort for developing an instance from scratch, developing a template-based reuse component, and assembling this reuse component.

i. Commention of the component designed w/o reuse.
ii. Documentation is a user manual.

The time for an experienced designer to develop from scratch a single instance of a component is set to 100%. All other numbers are standardized to this number.

The effort to design-for-reuse a component is relatively high. This is mainly due to the time spent for analyzing
existing solutions and extracting the requirements for a reuse component as well as due to the time spent for creating the documentation. On top of the functional description of the component, the documentation gives advice on how to refine the specification and how to transform the refined specification into VHDL. The implementation of the model is more complex because of these additional features, and also the effort for testing rises.

On the other hand, using such a reuse component takes much less effort than designing a model from scratch. The time to work into the subject decreases, the designer only has to learn how to use the components. Using the reuse component also reduces the time for documentation, since the designer can use the refined specification and generated reports for this purpose.

As a thumb rule, the overhead to develop a reuse component pays off at the fifth re-use.

5 Conclusion

We have presented a reuse scenario consisting of a library of reuse components i.e. extremely flexible reusable components together with the corresponding design process. The key elements of a reuse component are the highly flexible and self-generating VHDL models, the verification support, and the HyperText documentation. Our reuse scenario is currently applied in a large Siemens design center. First results prove the feasibility and the potential of our approach, and show, that the overhead of developing a reuse component grows off at the fifth re-use.

In contrast to other approaches, we built in design expertise throughout the complete design process in a way which allows us to reuse know-how. This also means that the creation of well-designed documentations has become crucial. We recognized that 20% to 30% of the effort of developing a reuse component is taken for creating the documentation.

The following problems were recognized with design reuse in general and with our approach in particular:

• Design reuse still must be committed to, both by management as well as by designers.
• Design partitioning and the definition of the encapsulated reuse component’s functionality must be done very carefully. Otherwise the development time for the reuse components increases drastically.
• Most of the tools do not support all VHDL concepts needed for reuse modeling. Hence time-consuming and error-prone work-arounds cannot be avoided.

Extending design reuse towards know-how reuse has proved to be a good strategy. Therefore, our scenario can be extended towards the specification phase in the following ways: First of all, the specification refinement procedure can be automated and provided to the specification rather than to the design teams. Second, reuse techniques should be established at the specification level by referring to reuse components and other standards from the very beginning and by allowing for reuse including other specification formats than VHDL, both executable and non-executable.

Acknowledgments

We would like to thank Dieter Zerweck, Rüdiger Hoffmann and Peter Picard for supporting the application of our reuse scenario in several ASIC projects, and Sigrid Junold and Harald Preiß for useful hints and suggestions.

References

[DW94] SYNOPSYS, DesignWare™ Manuals
[SYNO] SYNOPSYS, Design Compiler™ Manuals
[Tools] VSS / SYNOPSYS Sys1076/ Mentor Graphics Corp Voyager/IKOS