Transformative Hardware Design Following the Model-Driven Architecture Vision

Zhao Han[1,2](✉), Gabriel Rutsch[1], Deyan Wang[1,2], Bowen Li[1,2], Sebastian Siegfried Prebeck[1,2], Daniela Sanchez Lopera[1,2], Keerthikumara Devarajegowda[1], and Wolfgang Ecker[1,2]

[1] Infineon Technologies AG, Am Campeon 1-15, 85579 Neubiberg, Germany
{Zhao.Han,Gabriel.Rutsch,Deyan.Wang,Bowen.Li,Sebastian.Prebeck, Daniela.Lopera,Keerthikumara.Devarajegowda,Wolfgang.Ecker}@infineon.com
[2] Technical University Munich, Arcisstraße 21, 80333 Munich, Germany

Abstract. Despite the high configurability of IPs and hardware generators, code modifications are still required to introduce aspect-oriented instrumentation to satisfy emerging aspectual design requirements such as on-chip debug and functional safety. These code modifications escalate development, verification efforts, and deteriorate code reuse. This paper proposes a highly efficient transformative hardware design methodology that leverages graph-grammar-based model transformations. Following the proposed methodology, main design functionalities and aspectual instrumentation are separately developed, automatically integrated, and verified. To demonstrate the applicability, industrial SoCs were transformed to support on-chip debug. Compared to the manual RTL coding, the proposed transformative methodology needed less than 32x Lines of Code (LoC) to develop and integrate the aspectual instrumentation. In particular, our approach enables high code reusability, as the implementation of the transformation script is a one-time effort, and can be applied to all evaluated SoCs. This high LoC gain and code reuse promote the overall productivity of digital design.

Keywords: Electronic design automation · Aspect-oriented programming · Model-driven architecture

1 Introduction

With growing complexity in System on Chips (SoCs), the hardware development cycle is prolonged and the cost increases. Intellectual Property (IP) reuse is a major productivity booster in hardware development and helps to promote code reuse. Following the IP reuse methodology, designers are encouraged to build configurable IPs that encapsulate verified design implementations. After, IPs are adapted and integrated to build large and complex designs to accelerate the development cycle. For further code reuse, hardware generators are built to encode design knowledge with hardware generation frameworks [1–5]. Hardware generators enable design customization reuse and are implemented with high-level programming languages such as Scala and Python. Making use of modern programming paradigms such as object orientation, hardware generators can adapt highly complex configurations and generate adequate Hardware Description Language (HDL) code.

However, beyond main functionalities, aspectual design requirements such as On-Chip Debug (OCD) [6] and functional safety [7] have emerged over the years and are essential for product success. On-chip instrumentation satisfying these design requirements is dependent on core functionality realizations and demands system-wide support. Towards this end, code pieces are scattered across different IPs to implement the required on-chip instrumentation. For example, to support OCD, not only debug IPs (e.g. JTAG) but also special features are needed in existing IPs, e.g. hardware breakpoints in the CPU. But due to the absence of aspect orientation [8] in state-of-the-art HDLs (e.g. SystemVerilog [9]) and hardware generation frameworks (e.g. Chisel [2]), the OCD instrumentation is either always implemented [10] or configurable by increasing the IP generality [11]. The former option results in an additional chip area and introduces possible security breaches, whereas escalated development and verification efforts are expected in the latter one [12]. In this paper, we use the term *design aspects* to describe these scattered and tangled aspect-oriented on-chip instrumentation. Design aspects pose new challenges in hardware development, because the scattering and tangling make the hardware implementations hard to understand, maintain, and reuse.

To address these issues, we propose to weave aspect-oriented instrumentation by transforming existing designs leveraging graph grammar [13]. The proposed transformative hardware design methodology is built on top of a model-driven hardware generation framework, which follows the Model-Driven Architecture® (MDA®) vision [14]. The intermediate layer of this framework contains platform-independent design models that capture intended microarchitectures. Design models are graph-based: Hierarchical and logic components are vertices, whereas connections among ports and hierarchizations of components are edges. The proposed methodology transforms existing design models to incorporate desired aspect-oriented instrumentation. Model transformations are formalized and guided by graph grammar. Thus, the main contributions of this paper are: (1) Main functionalities and design aspects are decoupled and addressed separately with the proposed methodology. This separation of design concerns promotes modularity, reduces development efforts, and enables high code reuse. (2) For quality assurance, design constraints are developed to validate introduced design modifications. Besides, formal properties are automated to verify transformed designs. (3) One Domain-Specific Language (DSL) is used to construct and transform designs. This consistent design environment prevents semantic gaps and lowers integration and maintenance burdens.

The rest of the paper is organized as follows: Related work is discussed in the next section. Section 3 depicts the underlying model-driven hardware generation framework. After, the proposed transformative hardware design methodology is elaborated in Sect. 4. To demonstrate the applicability, Sect. 5 presents an industrial case study on RISC-V OCD transformation and discusses the experimental results. The last section concludes the significance of this paper.

2 Related Work

High-level languages such as SystemC are used to describe hardware and compiled to HDL to improve design productivity [15]. To further reuse design customizations, hardware generation frameworks such as Genesis2 [1] are proposed. However, they fail to separate design aspects from main functionalities. Designers must consider aspect-oriented instrumentation during the implementation of the main functionality. Consequently, the scattered and tangled-up aspect-oriented instrumentation leads to increased development and verification efforts [12].

Moreover, the intermediate layer of the hardware generation framework used by Chisel is called Flexible Intermediate Representation for RTL (FIRRTL) [3]. FIRRTL enables instrumentation insertion by rewriting its abstract syntax tree. In doing so, simple circuits such as hardware counters can be inserted into designs. Also, PyRTL [4] and PyMTL [5] follow the same idea to enable instrumentation transformation. Furthermore, FTI [16] provides a graphical user interface to assist hardware engineers to harden a design step by step. Internally, FTI translates designs written in VHDL to tree-based AIRE-CE representations [17].

However, there are three main drawbacks of these approaches. First, the underlying tree-based data structure is inappropriate, as it is not the intrinsic structure of circuits, i.e., graphs. As a result, development efforts escalate, which diminishes the gained design productivity. Second, the employment of different languages for design construction and transformation introduces semantic gaps and complicates hardware development. Most importantly, they do not generate verification artifacts to assure the quality of transformed designs.

3 Model-Driven Hardware Generation Framework

MDA® [18] established itself as an important part of modern development processes over the last decades. Following MDA®, a model-driven hardware generation framework called MetaRTL has been developed to improve hardware design productivity [14]. With this design-centric framework, designers can focus on design intent instead of implementation details such as code formatting.

MetaRTL consists of three layers: *Things*, *Design*, and *View* (Fig. 1). The things layer captures specification items into the *Formal Specification Model*. For satisfying specifications, designs are constructed with the expressive MetaRTL DSL [19] and stored as platform-independent *Design Models*. Subsequently, *Target Code Models* are derived for different HDLs and technologies (e.g. FPGA) in the view layer. State-of-the-art HDLs such as VHDL [20], Verilog [21], and SystemVerilog [9] are supported as the possible generator outputs.

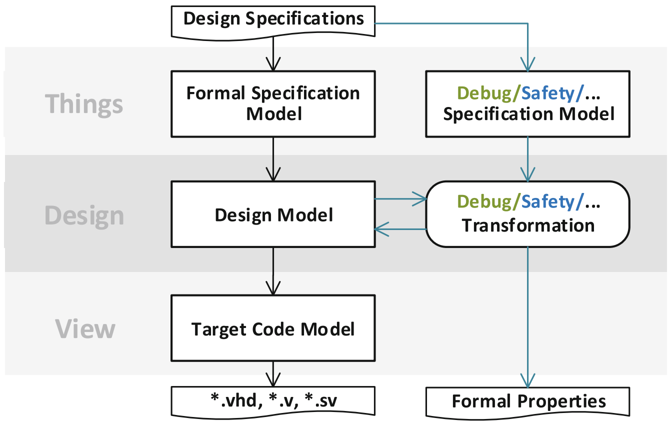

Fig. 1. Model-driven hardware generation framework (MetaRTL). The generation framework is extended with the transformative hardware design methodology as depicted in the right part.

A hardware generation flow starts from a formalized specification, which includes all must-have features and properties. In the main stage, i.e., design layer, the translation from the specification model to a design model is conducted by making use of a design template [22]. Design models are abstract RTL models and include all high-level design details, for instance, port connections among different logic gates. Next, the design model is translated to a target HDL code model by a view template [23], which includes target-specific aspects. Finally, the HDL code is derived from the target code model.

4 Transformative Hardware Design Methodology

The proposed methodology is implemented as part of the MetaRTL in Fig. 1. The left part in the figure shows the hardware generation flow, whereas the right part addresses design aspects with the transformative hardware design methodology. The proposed methodology integrates aspect-oriented instrumentation by transforming design models leveraging graph grammar. For easing hardware transformation development, various transformation utilities and reusable basic transformations are provided by the hardware transformation system. To assure quality, modified design models are validated with design constraints and transformed designs are verified with automated formal properties.

4.1 Graph-Based Design Model

In the main stage of MetaRTL, platform-independent design models capture microarchitectures. For describing microarchitectures, four types of components can be included in a design model [24]:

- *Descriptive* components indicate design hierarchies and description styles in the MetaRTL DSL [25].
- *Behavioural* components describe hardware behavior on a high level, such as finite state machines.
- *Sequential* components such as register store information, as their outputs depend on past and current inputs.
- *Primitive* components describe combinatorial logic, such as bitwise AND.

Hardware designs are usually represented as schematic diagrams, which use graphs to depict the instantiated components and connections among them. The design model is graph-based as well. Let LAB be an arbitrary but fixed set of suitable labels. A design model can be formalized with a hierarchical port graph [24]:

$$H = (V, P, E, (s_i, t_i)_{i \in \{G,T\}}, \mathfrak{p}, \mathfrak{t}, \mathfrak{d}, l)$$

- V is a finite set of vertices (components).
- P is a finite set of ports.
- $E = E_G \cup E_T$ are finite sets of graph and tree edges.
- s_G, s_T are source functions for graph and tree edges respectively.
- t_G, t_T are target functions for graph and tree edges respectively.
- $\mathfrak{p} : P \to V$ is a parent function that returns the parent component of a port.
- $\mathfrak{t} : V \to \mathit{Descriptive} \cup \mathit{Behavioural} \cup \mathit{Sequential} \cup \mathit{Primitive}$ is the type function for vertices (components).
- $\mathfrak{d} : P \to \{\mathit{In}, \mathit{Out}, \mathit{Inout}\}$ is the direction function for ports.
- $l : V \cup P \to LAB$ is a labeling function for vertices and ports.

The hierarchy in design models is a tree spanning over the same set of vertices V. This means, some vertices in design models are subgraphs that contain other vertices. These vertices are descriptive components, since they indicate design hierarchies and contain sub-components in design models. Hence, a vertex $v \in V$ can be denoted as

$$v = (V_v, P_v, E_v, (s_{i.v}, t_{i.v})_{i \in \{G,T\}}, \mathfrak{p}_v, \mathfrak{t}_v, \mathfrak{d}_v, l_v)$$

It is noteworthy that $V_v \subseteq V$, $E_V \subseteq E_G$, and $P_V \subseteq P$. Besides, these functions are restricted by the functions of the graph. This means, the global functions apply to subgraphs as well, i.e.,

$$v = (V_v, P_v, E_v, (s_i, t_i)_{i \in \{G,T\}}, \mathfrak{p}, \mathfrak{t}, \mathfrak{d}, l)$$

When a vertex $v \in V$ is not a descriptive component, then it does not contain any sub-component or edge, i.e., $V_v = v, E_v = \emptyset$. In contrast, when a vertex $v \in V$ is the top-level component, then $V_v = V, P_v = P$, and $E_v = E$. The top-level component is not a target of any tree edge.

For an illustrative example, the hierarchical port graph of a half adder is shown in Fig. 2. Labels of components and ports are illustrated, e.g. $l(v_0) =$ "*halfAdder*". The descriptive component v_0 indicates a hierarchy, i.e.,

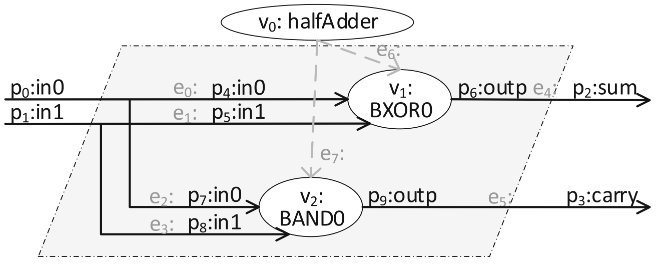

Fig. 2. Hierarchical port graph for a half adder. Solid black edges describe connections among ports and dashed grey edges depict hierarchizations of vertices (components). (Color figure online)

$\mathfrak{t}(v_0) \in \mathit{Descriptive}$. Where $V_{v_0} = \{v_1, v_2\}$, $P_{v_0} = \{p_0, p_1, ..., p_9\}$ and $E_{v_0} = \{e_0, e_1, ..., e_7\}$. Components v_1, v_2 are bitwise XOR and AND respectively.

Ports belong to components, i.e., $\mathfrak{p}(p_i)_{i\in\{0,1,2,3\}} = v_0$, $\mathfrak{p}(p_i)_{i\in\{4,5,6\}} = v_1$, and $\mathfrak{p}(p_i)_{i\in\{7,8,9\}} = v_2$. Directions of ports are: $\mathfrak{d}(p_i)_{i\in\{0,1,4,5,7,8\}} = \mathit{In}$ and $\mathfrak{d}(p_i)_{i\in\{2,3,6,9\}} = \mathit{Out}$. Connections among ports are identified by their source and target ports in the graph:

$$\begin{aligned}
s_G(e_0) &= p_0, t_G(e_0) = p_4\\
s_G(e_1) &= p_1, t_G(e_1) = p_5\\
s_G(e_2) &= p_0, t_G(e_2) = p_7\\
s_G(e_3) &= p_1, t_G(e_3) = p_8\\
s_G(e_4) &= p_6, t_G(e_4) = p_2\\
s_G(e_5) &= p_9, t_G(e_5) = p_3
\end{aligned}$$

In a design model, a port might be connected to multiple ports. But connections across hierarchies are forbidden. Therefore, a port such as p_4 can only be connected to another port that belongs to the current hierarchy (e.g. p_0) or a component in the current hierarchy (e.g. p_9). The port direction must be considered as well. A connection between two ports $p_i, p_j \in P$ is represented as $e_G \in E_G$, e.g. $e_0 = (p_0, p_4)$. Whilst, a tree edge $e_T \in E_T$ describes the hierarchical inclusion between two components $v_i, v_j \in V$, e.g. $e_7 = (v_0, v_2)$.

The uniqueness of labels is only assured inside a design hierarchy and among ports of the same component. Therefore, labels maybe not unique in a design model, e.g. $l(p_4) = l(p_7)$. To locate an exact component, additional hierarchical information is required. To this end, let $\mathfrak{h}$ be a *hierarchical path* from v_i to v_j, where $\mathfrak{t}(v_i) \in \mathit{Descriptive}$. A hierarchical path $\mathfrak{h}$ is a sequence of vertices $v_i, v_{i+1}, ..., v_j$ such that, for any $x = i, i+1, ..., j$, there exists an edge $e_T \in E_T$, that $s_T(e_T) = v_x$ and $t_T(e_T) = v_{x+1}$. That is, the previous descriptive component contains the next component in a hierarchical path.

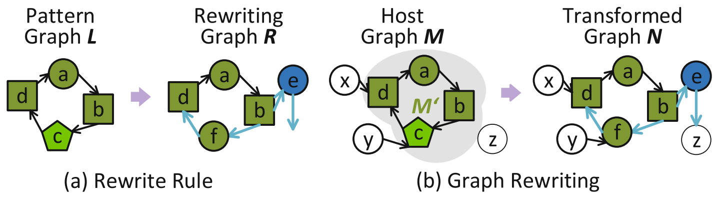

Fig. 3. Graph grammar. (a) A rewrite rule consists of a pattern graph L and a rewriting graph R. (b) By matching the subgraph M' in host graph M according to the pattern graph L, design modifications indicated by the rewriting graph R are inserted automatically to form the transformed graph N.

4.2 Design Model Transformation

The design model transformation can be referred to as graph rewriting. A graph rewriting is guided with graph grammar, which describes an iterative process of applying a set of rewrite rules on the matched subgraphs in the host graph [26]. For this purpose, the definition of graph morphisms and subgraphs is adapted for hierarchical port graphs.

Let M and N be two hierarchical port graphs. A *hierarchical port graph morphism* $\mathfrak{f}$ from M to N consists of three functions: $\mathfrak{f}_V, \mathfrak{f}_P$, and $\mathfrak{f}_E$. The interrelations of edges, hierarchical inclusions between components, labels and types of components, labels and directions of ports are preserved, i.e., for any $v_M \in V_M$, $p_M \in P_M$, $e_{M.G} \in E_{M.G}$, and $e_{M.T} \in E_{M.T}$, the following properties hold:

$$\begin{aligned}
&\mathfrak{f}_P(s_{M.G}(e_{M.G})) = s_{N.G}(\mathfrak{f}_E(e_{M.G}))\\
&\mathfrak{f}_P(t_{M.G}(e_{M.G})) = t_{N.G}(\mathfrak{f}_E(e_{M.G}))\\
&\mathfrak{f}_V(s_{M.T}(e_{M.T})) = s_{N.T}(\mathfrak{f}_E(e_{M.T}))\\
&\mathfrak{f}_V(t_{M.T}(e_{M.T})) = t_{N.T}(\mathfrak{f}_E(e_{M.T}))\\
&l(v_M) = l(\mathfrak{f}_V(v_M)) \wedge \mathfrak{t}(v_M) = \mathfrak{t}(\mathfrak{f}_V(v_M))\\
&l(p_M) = l(\mathfrak{f}_P(p_M)) \wedge \mathfrak{d}(p_M) = \mathfrak{d}(\mathfrak{f}_P(p_M))
\end{aligned}$$

Further, let M be a *subgraph* of a hierarchical port graph N, denoted as $M \subseteq N$, where $V_M \subseteq V_N$, $P_M \subseteq P_N$ and $E_M \subseteq E_N$, and the functions of M are restrictions of those in N. For any $v \in V_M$, if $\mathfrak{t}(v) \in$ *Descriptive*, then all subcomponents and edges in this descriptive component in N should be included in M as well, i.e., $V_{N.v} \subseteq V_M$, $P_{N.v} \subseteq P_M$, and $E_{N.v} \subseteq E_M$.

Algorithm 1 describes the graph-grammar-based design model transformation. A graph grammar consists of a set of rewrite rules $\mathfrak{R}$. A rewrite rule $r \in \mathfrak{R}$ consists of two graphs: left-hand-side pattern graph L and right-hand-side rewriting graph R (Fig. 3a).

The application of a rewrite rule consists of three steps:

1. *Match* a subgraph M' in host graph M that has a graph morphism from pattern graph L to M' (Fig. 3b).

Algorithm 1: Design Model Transformation

```
Input  : Host graph $M = (V_M, P_M, E_M, (s_{M.i}, t_{M.i})_{i\in\{G,T\}}, \mathfrak{p}_M, \mathfrak{t}_M, \mathfrak{d}_M, l_M)$,
         Rules $\mathfrak{R} = \{r_0, r_n, ..., r_n\}$
Output: Transformed Graph $N$
Let $N$ be the duplication of $M$
for $r = (L, R) \in \mathfrak{R}$ do
    $M' = \mathsf{match}(M, L)$
    for $v \in V_L$ and $v \notin V_R$ do
        $V_N = V_N \setminus \{v\}$
    end
    for $e \in E_L$ and $e \notin E_R$ do
        $E_N = E_N \setminus \{e\}$
    end
    for $v \in V_R$ and $v \notin V_L$ do
        $V_N = V_N \cup \{v\}$
    end
    for $e \in E_R$ and $e \notin E_L$ do
        $E_N = E_N \cup \{e\}$
    end
end
return $N = (V_N, P_N, E_N, (s_{N.i}, t_{N.i})_{i\in\{G,T\}}, \mathfrak{p}_N, \mathfrak{t}_N, \mathfrak{d}_N, l_N)$
```

2. *Remove* the components, ports and edges that belong to L but not R.
3. *Add* the components, ports and edges that belong to R but not L.

In the algorithm, the function $\mathsf{match}(M, L)$ returns the found subgraph M'. During transformations, graph functions such as labeling function l are updated automatically when the host graph is modified.

To find the subgraph M' in host graph M, components in M are located for every vertex in L with its hierarchical path, name, and type. Since the name and direction of ports are preserved in the matched subgraph, connections are recognized by locating the linked ports. The located components, ports, and edges compose the matched subgraph M'. In doing so, a graph morphism from M' to L is derived. Based on such graph morphism, design modifications described by the rewriting graph R are incorporated automatically to form the transformed graph N.

Design construction is a special case of design transformation, where the pattern graph L is always a subgraph of the rewriting graph R in rewrite rules. That is, the removal of components, ports, and connections is absent in design construction. Inspired by this observation, we use the MetaRTL DSL to not only construct but also transform designs.

4.3 Hardware Transformation System

To ease hardware transformation development, a hardware transformation system is developed (Fig. 4). Essential operations such as *Localization*, *Removal*, and *Addition* of components, ports, and connections are needed during design model transformations. These operations are performed with the *Transformation Utilities*. Making use of transformation utilities, various *Transformations* are developed. Transformations are classified into *Basic*, *Design-Independent*,

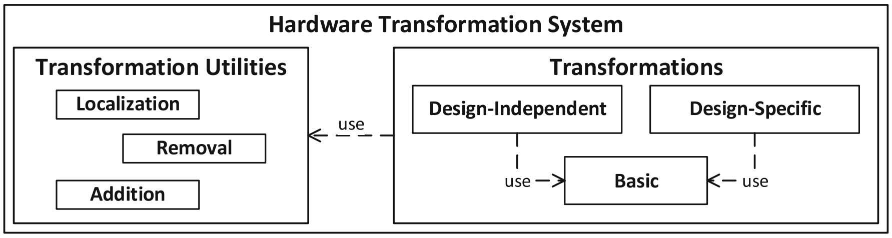

Fig. 4. Hardware transformation system. Transformation utilities consist of localization, removal, and addition utilities for components, ports, and connections. Making use of these utilities, basic transformations introduce elementary design modifications and are reused to construct complex design-independent and design-specific transformations.

and *Design-Specific* transformations. A basic transformation introduces an elementary design modification, which affects only a few components, ports, and connections in design models. In contrast, by reusing basic transformations, complex design-independent and design-specific transformations are developed with reduced efforts for systematic design modifications. These systematic transformations differ due to microarchitectural dependency.

Transformation Utilities. During hardware transformation, specific components, ports, and connections are modified to introduce design modifications. For this purpose, hardware transformations start with the localization of target components, ports, and connections. This step is formalized as the first application step of rewrite rules. To assist this step, the MetaRTL DSL offers localization utilities. For example, a simplified component localization function is shown in Listing 1.1.

```
def componentLocalization(hierarchy, name, comp_type, path, designModel)
    :
  founds = list()
  for comp in hierarchy.getComponents():
    currentHierarchicalPath = getHierarchicalPath(comp, designModel)
    if currentHierarchicalPath in path:
      if isinstance(comp, Descriptive):
        foundComps = componentLocalization(comp, name, comp_type, path,
    designModel)
        founds.extends(foundComps)
    elif currentHierarchicalPath == path:
      if comp.getName() == name:
        if isinstance(comp, comp_type):
          founds.append(comp)
    continue
  return founds
```

Listing 1.1. Component Localization Function. This function localizes a component based on its name, type, and hierarchical path. In doing so, adequate components in the design model are located and returned as a list.

The shown component localization function has five arguments: the current design *hierarchy*, target component *name*, *comp_type*, its absolute hierarchical *path*, and the *designModel*. First, components under the current design hierarchy are iterated in line 3. The absolute hierarchical path of the iterated component (*comp*) is retrieved with the *getHierarchicalPath* function in line 4. By comparing the retrieved hierarchical path (*currentHierarchicalPath*) to the given path, different steps are followed: If the retrieved hierarchical path is part of the given absolute hierarchical path and the iterated component is a descriptive component (lines 5–6), target components are located in the design hierarchy indicated by the iterated component. Then, the localization process conducts further inside the iterated component in line 7. But if the retrieved hierarchical path is identical to the given absolute hierarchical path, component name and type are then compared (lines 9–11). When all these search criteria are satisfied, the current iterated component is marked in line 12. However, if the retrieved hierarchical path does not satisfy the previous conditions, the component localization function continues to iterate the next component in the current design hierarchy.

After the localization step in model transformations, components, ports, and/or connections are removed from and/or added in located subgraphs to introduce design modifications. During this process, three graph-based operations are observed: remove, add, and replace. The remove and add operations are supported inherently by the MetaRTL DSL [19], while the replace operation is the composition of remove and add operations. For example, an exemplary component replacement function is shown in Listing 1.2.

```
 1 def componentReplacement(original, new):
 2   hierarchy = original.parent
 3   for port in original.getPorts():
 4     newPort = new.getPort(Name=port.getName())
 5     connections = getConnections(port, hierarchy)
 6     for connection in connections:
 7       connection.delConnector(port)
 8       connection.addConnector(newPort)
 9   hierarchy.delComponent(original)
10   hierarchy.insComponent(new)
```

Listing 1.2. Component replacement function. This simplified function considers only components with identical port definitions in terms of name, type, and number.

The shown component replacement function replaces the *original* component with the *new* component. The target design *hierarchy* is located in line 2. Since both components have identical port definitions, connections linking the ports of the original component are rewired in lines 3–8. To do this, the port *newPort* of the new component is retrieved with the *getPort* function by the name of the iterated *port* in line 4. The *connections* linking the iterated *port* under the current design hierarchy are then obtained with the *getConnections* function in line 5. Later, these connections are rewired by replacing the iterated port with the *newPort* (lines 6–8). Afterward, the original component is removed and the new component is placed under the same design hierarchy (lines 9–10).

Moreover, design details such as the related connections and connected ports of the target port are often required in model transformations. For this purpose, transformation utilities for connections such as *getConnections* in Listing 1.2 are provided. Various transformation utilities are served as the intuitive programming interface for hardware transformation and are the fundament of the hardware transformation system.

Basic Transformations. Basic transformations introduce elementary design modifications into design models. Exemplary basic transformations are the naming convention transformation and safety mechanism transformations.

A basic transformation for the naming convention has been developed [25] to adapt component and port names to different design projects. That is, specific prefixes and suffixes are often predefined for different design elements to avoid ambiguity and assist readability. For example, ports are named with the prefix "p_". Also, the name suffix "_i" or "_o" of ports indicates the direction. With this naming convention transformation, component and port names are adapted automatically to meet coding guidelines in the target design project.

Moreover, the transformation system offers basic transformations for various safety mechanisms [27]. Safety mechanisms introduce redundancy into the system to enable error detection (and correction). In doing so, the system is maintained in a safe state and, thus, dangerous consequences caused by malfunctions are reduced. The introduced redundancy has three categories: information redundancy, hardware redundancy, and time redundancy [7]. For introducing information redundancy into the hardware system, basic transformations are provided for parity error detection code, CRC, hamming code, etc. Whilst, basic transformations for hardware redundancy mechanisms such as Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR) are also provided. But time redundancy safety mechanisms are software-based [28] and, thus, not offered as basic transformations.

Basic transformations are implemented in a configurable and modular manner. Hereby, basic transformations can serve as building blocks and form the "transformation library". This modularity and reusability assist and ease the complex hardware transformation development.

Design-Specific/-Independent Transformations. To address aspectual design requirements such as on-chip debug [6] and functional safety [7], design-specific and -independent transformations introduce systematic design modifications to design models. As the name indicates, design-specific transformations are dependent on the microarchitecture and, thus, are applicable to only a set of designs. Whereas, design-independent transformations have no such restrictions and are applicable to all designs.

Design-specific transformations are highly dependent on the microarchitecture. For example, the RISC-V OCD transformation introduces on-chip debug support in a CPU subsystem that implements RISC-V ISA [29]. In this ISA specification, the exception handling behavior and related information storage are detailed for RISC-V architecture. If the target architecture does not support the RISC-V ISA, these design details may differ and, thus, the RISC-V OCD transformation is not applicable anymore.

In contrast, design-independent transformations are independent of the microarchitecture. For example, a systematic functional safety transformation has been developed [27]. This functional safety transformation reuses basic transformations for safety mechanisms to harden sequential components such as registers and memories in a design. Moreover, different design projects may have different coding requirements for e.g. linting checks and code review. Thus, different RTL coding styles may be required. For this purpose, design-independent model transformations have been developed to fine-tune design models to vary IP-coding styles [25].

4.4 Quality Assurance

Model Validation. After transformations, modified design models are validated against design constraints. This validation assures the consistency of design models, which ensures that the generated HDL code is synthesizable. Following design constraints must be satisfied by all design models:

– *Multiple Connections*: For any two connections $e_i, e_j \in E_G$, their source and target can not be identical at the same time.

$$\begin{aligned} &s_G(e_i) = s_G(e_j) \wedge t_G(e_i) \neq t_G(e_j) \vee \\ &s_G(e_i) \neq s_G(e_j) \wedge t_G(e_i) = t_G(e_j) \vee \\ &s_G(e_i) \neq s_G(e_j) \wedge t_G(e_i) \neq t_G(e_j) \end{aligned}$$

– *Cross-Hierarchy Connections*: Connections across hierarchies are not allowed. To simplify the notation, we introduce a helper function $\mathfrak{s} : V \rightarrow V$ that returns the parent component of a component. For any connection $e \in E_G$, its source and target port must belong to the same hierarchy, i.e.,
  - if $\mathfrak{t}(\mathfrak{p}(s_G(e))), \mathfrak{t}(\mathfrak{p}(t_G(e))) \notin \mathit{Descriptive}$, then

$$\mathfrak{s}(\mathfrak{p}(s_G(e))) = \mathfrak{s}(\mathfrak{p}(t_G(e)))$$

  - if $\mathfrak{t}(\mathfrak{p}(s_G(e))) \in \mathit{Descriptive}$ and $\mathfrak{t}(\mathfrak{p}(t_G(e))) \notin \mathit{Descriptive}$, then

$$\begin{aligned} &\mathfrak{s}(\mathfrak{p}(s_G(e))) = \mathfrak{s}(\mathfrak{p}(t_G(e))) \vee \\ &\mathfrak{p}(s_G(e)) = \mathfrak{s}(\mathfrak{p}(t_G(e))) \end{aligned}$$

  - if $\mathfrak{t}(\mathfrak{p}(s_G(e))), \mathfrak{t}(\mathfrak{p}(t_G(e))) \in \mathit{Descriptive}$, then

$$\begin{aligned} &\mathfrak{s}(\mathfrak{p}(s_G(e))) = \mathfrak{s}(\mathfrak{p}(t_G(e))) \vee \\ &\mathfrak{p}(s_G(e)) = \mathfrak{s}(\mathfrak{p}(t_G(e))) \vee \\ &\mathfrak{s}(\mathfrak{p}(s_G(e))) = \mathfrak{p}(t_G(e)) \end{aligned}$$

– *Dangling Connections*: For any connection $e \in E_G$, there must exist a source and a target port, i.e.,

$$s_G(e), t_G(e) \in P$$

- *Valid Connections*: For any connection $e \in E_G$, it must be feasible in terms of port directions. As an exception, a connection is always feasible, if any of the connected ports has the direction "Inout". Thus, following constraints apply to a connection $e \in E_G$ that connects ports with direction either "In" or "Out", i.e., $\mathfrak{d}(s_G(e)), \mathfrak{d}(t_G(e)) \in \{In, Out\}$.
  - If two connected ports belong to components under the same hierarchy, i.e., $\mathfrak{s}(\mathfrak{p}(s_G(e))) = \mathfrak{s}(\mathfrak{p}(t_G(e)))$, then connector directions must differ.

$$\mathfrak{d}(s_G(e)) \neq \mathfrak{d}(t_G(e))$$

  - If two connected ports belong to components under different hierarchies, i.e., $\mathfrak{s}(\mathfrak{p}(s_G(e))) \neq \mathfrak{s}(\mathfrak{p}(t_G(e)))$. This means, one of these components is a descriptive component that contains the other connector's parent, then connector directions must be identical.

$$\mathfrak{d}(s_G(e)) = \mathfrak{d}(t_G(e))$$

- *Zero-Driven Connections*: Except input ports of the top component, for any port $p \in P$, there exists $e \in E_G$ such that

$$t_G(e) = p$$

- *Multi-Driven Connections*: For any two connections $e_i, e_j \in E_G$, they must have different targets.

$$t_G(e_i) \neq t_G(e_j)$$

- *Unconnected Component*: For any port $p \in P$, it must be connected, i.e., there exists a connection $e \in E_G$ such that

$$s_G(e) = p \vee t_G(e) = p$$

- *Single Hierarchy*: For any component $v \in V$, it must be located in only one hierarchy. This means, for any two hierarchical inclusions $e_i, e_j \in E_T$, their targets must be different.

$$t_T(e_i) \neq t_T(e_j)$$

Design Verification. To verify the design functionality, two verification suites are employed: existing regressions tests for main functionalities and newly automated formal properties for introduced design modifications.

Existing regression tests consisting of verification artifacts such as testbenches are developed for existing designs. Because the proposed approach targets design aspects, the main functionalities should stay intact. Thus, with adequate additional constraining, transformed designs must behave identically as original designs in regression tests.

Further, a formal property generation framework is used to automate formal properties for introduced design modifications [30]. The meta-information produced by applying a rewrite rule indicates the component and connection modifications. With this information, the property templates developed from design specifications generate suitable formal properties. For example, a rewrite rule is developed to harden a General-Purpose Register (GPR) *sp* under the register file (*RF*) with DMR. The DMR inserts a duplicated register *sp_copy*. A new signal *err_det* indicates an error when the outputs of *sp* and *sp_copy* differ. Afterward, we apply this rule to a design model, where *RF* is in instruction decode stage *ID* in *CPU*. Alongside the rewrite rule application, formal properties are generated in SystemVerilog Assertions to verify the inserted DMR (Listing 1.3).

```
property RegisterFile_sp_DMR_ErrorFree ;
    ( CPU.ID.RF.sp.Out == CPU.ID.RF.sp_copy.Out)
    |->
    ( CPU.ID.RF.sp.err_det == 0 ) ;
endproperty
property RegisterFile_sp_DMR_ErrorDetection ;
    ( CPU.ID.RF.sp.Out != CPU.ID.RF.sp_copy.Out)
    |->
    ( CPU.ID.RF.sp.err_det == 1 ) ;
endproperty
```

Listing 1.3. Generated properties for DMR transformation. The first property verifies the error-free scenario, whereas the second property is for the erroneous scenario.

The proposed methodology complements the hardware generation. By rewriting existing design models following graph-grammar-based transformations, design aspects are addressed separately from main functionalities. This separation of design concerns reduces complexity in hardware generators, which can be developed and verified with decreased efforts. Moreover, the development of hardware transformations is assisted with the transformation utilities and reusable basic transformations provided by the hardware transformation system. After transformation, modified design models are validated by design constraints and formal properties are generated to verify introduced aspect-oriented instrumentation. In particular, because one DSL is used for design construction and transformation, the proposed approach avoids semantic gaps, lowers integration, and maintenance burdens.

5 Case Study

In this section, a case study is conducted on the RISC-V OCD transformation. To demonstrate the applicability, we apply the OCD transformation to different industrial SoCs. Resource utilization and the time to conduct hardware transformations are discussed. Subsequently, the achieved code reusability and required development efforts are analyzed.

5.1 RISC-V On-Chip Debug Automation

With the increasing complexity in chips and stringent time to market requirements, post-silicon firmware debug solutions such as In-Circuit Emulator (ICE) becomes rapidly unfavorable because of the high cost and long development time.

Whereas, with low-cost hardware probes, OCD instrumentation provides dedicated debug circuitry for a reasonable area increase. An OCD system provides advanced functionalities such as hardware breakpoints, single stepping, register, and memory accesses [6]. To support these OCD features, system-wide design modifications are required. For instance, the CPU needs to allow external register accesses and the bus matrix needs to support external memory accesses. It is also important to note that these design modifications are highly microarchitecture-specific. To separate these design concerns, the OCD automation is developed following the proposed transformative hardware design methodology.

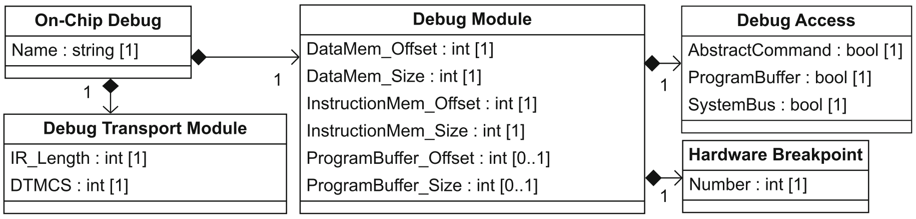

Fig. 5. RISC-V on-chip debug specification metamodel. This metamodel captures the main features of the debug transport module, debug module, and required system-wide support.

For formalizing the OCD requirements, the *On-Chip Debug* metamodel is abstracted from the RISC-V debug specification [6] (Fig. 5). Other than design-specific information such as the *Debug Transport Module* (DTM) instruction register length (*IR_Length*), memory offset and size, and the *Number* of supported *Hardware Breakpoints*, *Debug Accesses* can be configured as well. That is, the *Debug Module* (DM) may support different debug accesses: *AbstractCommand*, *ProgramBuffer*, and *SystemBus*. Since it is mandatory to support GPR accesses with the abstract command, these debug accesses differ in terms of Control and Status Registers (CSRs) and memory access methods. In this case study, abstract-command-based OCD implementation is used as an example.

The OCD transformation consists of basic transformations for enabling external register access, inserting hardware breakpoint CSRs, supporting breakpoint exception in the exception pipeline, etc. These basic transformations target different system parts and insert design modifications depending on the microarchitecture. In this paper, we focus on basic transformations for two essential design modifications: *register access* and *hardware breakpoint (HWBP)*.

The register access transformation adds several multiplexers for accessing the *GPR and CSR Units* (Fig. 6). The GPR unit supports two concurrent register read accesses. These registers are addressed by the *rs1_addr* and *rs2_addr*. Ports *rs1_data* and *rs2_data* indicate respective register data. The write access is supported by the *rd_addr*, *rd_data*, and *wr_en*. They carry the address, data, and write enable signals respectively. In contrast, the CSR unit supports only

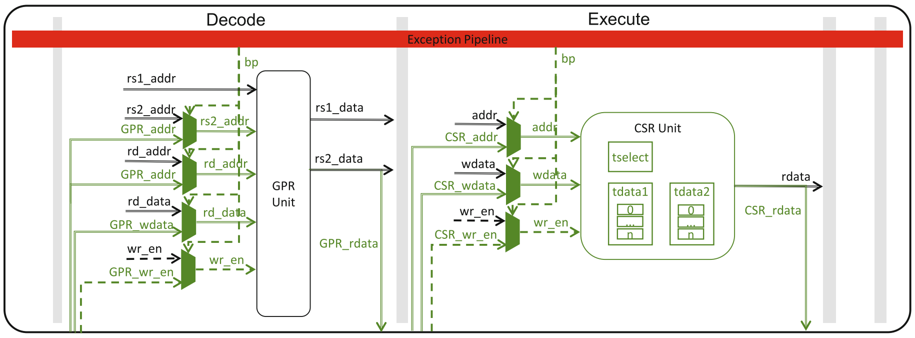

Fig. 6. On-chip debug transformation of RISC-V CPU. The CPU has a superscalar five-pipeline-stage architecture [31]. For simplicity, only transformation-related design details in the Decode and Execute stages are depicted.

one concurrent read or write access. Thus, the CSR unit has a much simpler interface: *addr*, *wdata*, *wr_en*, and *rdata*.

```
 1 def transformGPRUnit(designModel, debugModule):
 2   EP = componentLocalization(designModel, "ExceptionPipeline")
 3   GPRUnit = componentLocalization(designModel, "GPRUnit")
 4   bp_sig = EP.getPort(Name="bp")
 5   transform_dict = {"rs2_addr" : "GPR_addr",
 6   "rd_addr" : "GPR_addr",
 7   "rd_data" : "GPR_wdata",
 8   "wr_en" : "GPR_wr_en",
 9   "rs2_data" : "GPR_rdata"}
10   for orig_sig, debug_sig in transform.iteritems():
11     target_port = GPRUnit.getPort(Name=orig_sig)
12     mux = Mux(Name="{}_mux".format(orig_sig), Sel=bp_sig, parent=GPRUnit
       ._Parent)
13     connection = getConnection(target_port, GPRUnit._Parent)
14     connection.delConnector(target_port)
15     connection.addConnector(mux.addIn())
16     debug_port = debugModule.getPort(Name=debug_sig)
17     mux.addIn().connect(debug_port)
18   rs2_data = GPRUnit.getPort(Name="rs2_data")
19   GPR_rdata = debugModule.getPort(Name="GPR_rdata")
20   rs2_data.connect(GPR_rdata)
```

Listing 1.4. Register accesses transformation of the GPR unit. This transformation locates and rewrites the rs2 and rd of the GPR unit for the read and write access respectively.

The transformation that enables external GPR accesses is shown in Listing 1.4. In the *transformGPRUnit*, the exception pipeline (*EP*) and GPR unit (*GPRUnit*) are first located in lines 2–3. The signal *bp_sig* provided by the exception pipeline indicates whether the CPU enters debug mode or not (line 4). Signals such as *GPR_addr*, *GPR_wdata*, *GPR_wr_en*, and *GPR_rdata* belong to the *debugModule* and indicate the current debug GPR access. For allowing the external GPR accesses in CPU, a mapping of ports of the GPR unit and DM is defined in lines 5–9. Based on this port mapping, target input ports of the GPR unit are located and their connections are reworked (lines 10–17). In doing so, the target inputs of the GPR unit are re-connected to insert several multiplexers (*mux*). These multiplexers are controlled by the *bp_sig*. That is, when the CPU enters debug mode, the GPR unit is accessed by the DM, otherwise, by the CPU. After, the *rs2_data* signal is connected to the *GPR_rdata* (lines 18–19). Following similar steps, external CSR accesses are enabled.

Furthermore, the HWBP transformation transforms specific CSRs [6] in the *CSR unit*: *tselect*, *tdata1*, *tdata2* (Fig. 6). The tdata1 stores the configuration of an HWBP, whereas the tdata2 stores the comparison data, e.g. target instruction address. For supporting multiple HWBPs, the tdata1 and tdata2 are designed as virtual CSRs, where multiple CSRs are accessible with the same CSR address. The tselect CSR determines the current accessible HWBP CSRs.

```
 1 class TSELECT(Descriptive):
 2   def __init__(self, DebugMoT)
 3 class TDATA1(Descriptive):
 4   def __init__(self, DebugMoT)
 5 class TDATA2(Descriptive):
 6   def __init__(self, DebugMoT)
 7 def transformHWBP(designModel, DebugMoT):
 8   CSRUnit = componentLocalization(designModel, "CSRUnit")
 9   tselect = componentLocalization(CSRUnit, "tselect")
10   tdata1 = componentLocalization(CSRUnit, "tdata1")
11   tdata2 = componentLocalization(CSRUnit, "tdata2")
12   componentReplacement(tselect, TSELECT(DebugMoT))
13   componentReplacement(tdata1, TDATA1(DebugMoT))
14   componentReplacement(tdata2, TDATA2(DebugMoT))
```

Listing 1.5. Hardware breakpoint transformation of the CSR unit. This transformation is simplified, since it only considers the scenarios, when inadequate hardware breakpoint CSRs are already implemented.

Listing 1.5 illustrates the HWBP transformation in the CSR unit. In this transformation, the target design hierarchy (*CSRUnit*) and *tselect*, *tdata1*, and *tdata2* CSRs are located in lines 8–11. By replacing these original CSRs with the instantiated HWBP CSRs according to the *DebugMoT*, the HWBP transformation is complete (lines 12–14). The component replacement utility (*componentReplacement*) is part of transformation utilities.

Following the proposed methodology, the OCD transformation serves as the single source for implementing and integrating RISC-V OCD support. Subsequently, the transformed design models are validated against design constraints and the inserted OCD instrumentation is verified exhaustively with the automated formal properties.

5.2 Results

Five industrial 32-bit RISC-V SoCs with different feature sets targeting the powertrain market are evaluated. Besides peripherals such as Serial Peripheral Interface (SPI) and timers, the supported ISA extensions differ as well. Other than the base integer instruction set, SoCs may support standard extensions such as compressed instructions [29]. Additionally, customized multiply-accumulate instructions can be supported to boost the execution of machine learning applications. The more features are supported, the more complex is the design as shown in Fig. 7.

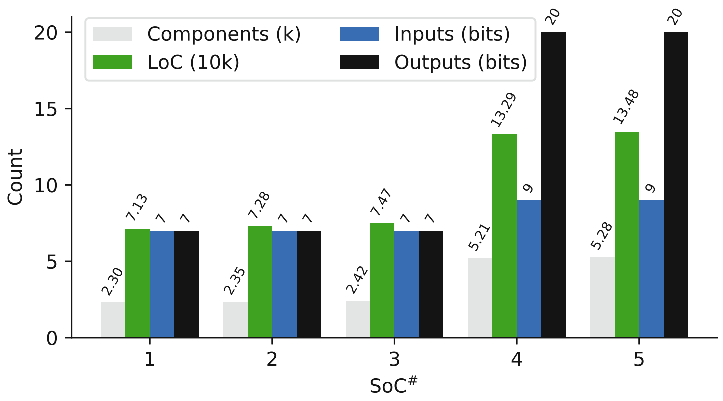

Fig. 7. Design complexity of experimented SoCs. The complexity is indicated by four factors: the number of components, LoC of the code base, bit width of input, and output ports.

Since the GPR and CSR units in evaluated SoCs have the same interface, the register access basic transformation introduces the same design modifications. To diversify the inserted design modifications, different amounts of HWBPs are enabled in SoCs, i.e., 4 HWBPs are enabled in SoC^1, SoC^2 and SoC^3, whereas SoC^4 and SoC^5 are enabled with 8 and 12 HWBPs respectively.

The resource utilization of SoCs is reported by the Vivado® v2018.1 design tool targeting the Arty-7 FPGA board from Xilinx®. All experiments are conducted on an Intel Xeon CPU E5-2690 v4 machine.

Design Area. Figure 8 shows the register and Look-Up-Table (LUT) utilization for the SoCs with and without OCD respectively. Resource utilization indicates the design area. A similar design area increase is observed in SoC^1, SoC^2 and SoC^3. With more HWBPs supported, SoC^4 and SoC^5 require more resources. The area penalty introduced by OCD cannot be neglected, which implies the importance of the RISC-V OCD automation.

Transformation Time. The time consumption of transforming an existing design model into a new design model is defined as the transformation time (Fig. 9). In general, more components and LoC of an SoC indicate a more complex graph representation in terms of vertices, ports, and edges. The increasing graph elements complicate the first step of the rewrite rule application, i.e., subgraph matching. However, around 1.1 s was used to enable external register access for SoCs with different design complexity. The reason is that the proposed approach can match a subgraph efficiently with hierarchical information.

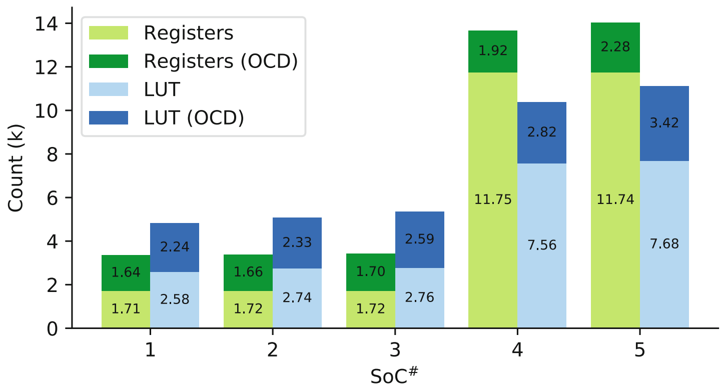

Fig. 8. Resource utilization before and after RISC-V On-Chip Debug (OCD) transformation. Light colors show the resource utilization of experimented SoCs, whereas dark colors depict utilization changes after RISC-V OCD transformation. (Color figure online)

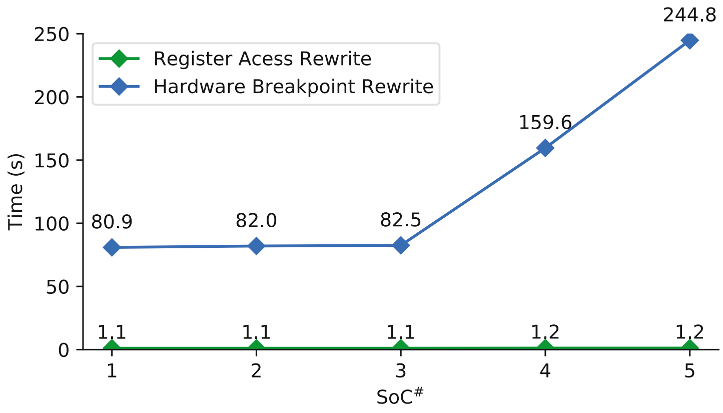

Fig. 9. Transformation time to apply rewrite rules. The hardware breakpoint rewrite rule introduces more design modifications when more hardware breakpoints are enabled. Whereas, the register access rewrite rule introduces the same design modifications in experimented SoCs.

Furthermore, a proportional relationship is observed between the number of enabled HWBPs and the transformation time. This shows, the transformation time for HWBPs increases linearly with introduced design modifications and is independent of the design complexity of target SoCs. This observation confirms the scalability and efficiency of the proposed methodology.

Development Efforts. Table 1 shows the required LoC for hardware transformations in the *MetaRTL* and state-of-the-art manual *VHDL* coding. Following the proposed transformative design methodology, 1.8k LoC is required to implement the RISC-V OCD automation. Whilst, at least 59k LoC needs to

Table 1. On-chip debug development efforts (LoC)

Platform	SoC[1]	SoC[2]	SoC[3]	SoC[4]	SoC[5]
VHDL	59.0k	60.4k	62.3k	66.5k	73.1k
MetaRTL	1.8k	–	–	–	–

be revisited to implement and integrate the RISC-V OCD instrumentation for evaluated SoCs with the manual approach. As a result, an LoC gain of more than 32x is observed. Further, the error-prone manual approach is replaced with the proposed design transformation, which promotes further modularity and automation in hardware development. Finally, it is important to note that the development efforts with MetaRTL are shown only for SoC[1] in the second row. This means, the transformation script requires only one-time implementation efforts and is applicable for all evaluated SoC models. The actual LoC gain and the code reusability factors are high.

6 Conclusion

In this paper, we propose to satisfy emerging aspectual design requirements such as On-Chip Debug (OCD) and functional safety with the transformative hardware design methodology. The proposed methodology is supported by graph-grammar-based model transformations that are implemented as part of a model-driven hardware generation framework. The model transformations enable aspect orientation in the conventional hardware generation. The aspect orientation separates design concerns and assures high modularity in hardware development. As a result, the complexity of hardware generators is reduced since their focal point is the core functionalities. Whilst, the aspect-oriented instrumentation is separately developed and automatically incorporated with model transformations. For easing transformation development, transformation utilities and reusable basic transformations are provided in the hardware transformation system. To assure quality, introduced design modifications are validated against design constraints and formal properties are generated to verify the transformed designs. Of note, we use one DSL to construct as well as transform designs, which prevents semantic gaps and lowers integration and maintenance burdens in hardware development. To demonstrate the applicability, the RISC-V OCD transformation was implemented and applied to different industry-strength SoCs. Compared to manual VHDL development, the LoC to develop and integrate the OCD instrumentation is reduced more than 32x with the proposed methodology. In particular, the transformation script requires only one-time implementation efforts and is applicable for different SoCs. The achieved high LoC gain and code reuse improve the overall productivity of digital design.

Acknowledgements. The work described herein is partly funded by the German Federal Ministry of Education and Research (BMBF) as part of the research project Scale4Edge (16ME0122K). It has its roots in the BMBF funded projects SAVE4I and COMPACT.

References

1. Shacham, O., et al.: Avoiding game over: bringing design to the next level. In: DAC Design Automation Conference 2012, pp. 623–629. IEEE (2012)
2. Bachrach, J., et al.: Chisel: constructing hardware in a scala embedded language. In: DAC Design Automation Conference 2012, pp. 1212–1221. IEEE (2012)
3. Izraelevitz, A., et al.: Reusability is FIRRTL ground: hardware construction languages, compiler frameworks, and transformations. In: Proceedings of the 36th International Conference on Computer-Aided Design, pp. 209–216. IEEE Press (2017)
4. Clow, J., Tzimpragos, G., Dangwal, D., Guo, S., McMahan, J., Sherwood, T.: A pythonic approach for rapid hardware prototyping and instrumentation. In: 2017 27th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–7. IEEE (2017)
5. Jiang, S., Pan, P., Ou, Y., Batten, C.: PyMTL3: a python framework for open-source hardware modeling, generation, simulation, and verification. IEEE Micro **40**(4), 58–66 (2020)
6. Tim, N., Megan, W.: RISC-V external debug support version 0.13.2. Technical report, SiFive Inc. (2019)
7. ISO, ISO26262: 26262: Road Vehicles-Functional Safety. International Standard ISO/FDIS, vol. 26262 (2011)
8. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
9. Marconi, S., Conti, E., Placidi, P., Christiansen, J., Hemperek, T.: IEEE standard for SystemVerilog-unified hardware design, specification, and verification language (2013)
10. Traber, A., et al.: PULPino: a small single-core RISC-V SoC. In: 3rd RISCV Workshop (2016)
11. Asanovic, K., et al.: The rocket chip generator. EECS Department, University of California, Berkeley, Technical report UCB/EECS-2016-17 (2016)
12. Gajski, D.D., Wu, A.C.-H., Chaiyakul, V., Mori, S., Nukiyama, T., Bricaud, P.: Embedded tutorial: essential issues for IP reuse. In: Proceedings of the 2000 Asia and South Pacific Design Automation Conference, pp. 37–42 (2000)
13. Ehrig, H., Habel, A., Kreowski, H.-J.: Introduction to graph grammars with applications to semantic networks. Comput. Math. Appl. **23**(6–9), 557–572 (1992)
14. Ecker, W., Devarajegowda, K., Werner, M., Han, Z., Servadei, L.: Embedded systems' automation following OMG's model driven architecture vision. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1301–1306. IEEE (2019)
15. Moiseev, M., Popov, R., Klotchkov, I.: SystemC-to-verilog compiler: a productivity-focused tool for hardware design in cycle-accurate SystemC. In: Design and Verification Conference and Exhibition (DVCon) Europe (2020)
16. López-Ongil, C., Entrena, L., García-Valderas, M., Portela-García, M.: Automatic tools for design hardening. In: Velazco, R., Fouillat, P., Reis, R. (eds.) Radiation Effects on Embedded Systems, pp. 183–200. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5646-8_9
17. Reshadi, M.H., Gharehbaghi, A.M., Navabi, Z.: AIRE/CE: a revision towards CAD tool integration. In: ICM 2000. Proceedings of the 12th International Conference on Microelectronics. (IEEE Cat. No. 00EX453), pp. 277–280 (2000)

18. Kleppe, A.G., Warmer, J., Warmer, J.B., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional (2003)
19. Han, Z., Devarajegowda, K., Werner, M., Ecker, W.: Towards a python-based one language ecosystem for embedded systems automation. In: 2019 IEEE Nordic Circuits and Systems Conference (NorCAS): NORCHIP and International Symposium of System-on-Chip (SoC), pp. 1–7. IEEE (2019)
20. I. Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE Standard VHDL Language Reference Manual. IEEE (2000)
21. Sutherland, S.: The IEEE Verilog 1364–2001 standard what's new, and why you need it. In: 9th International HDL Conference (HDLCon) (2000)
22. Schreiner, J., Findenigy, R., Ecker, W.: Design centric modeling of digital hardware. In: 2016 IEEE International High-Level Design Validation and Test Workshop (HLDVT), pp. 46–52. IEEE (2016)
23. Schreiner, J., Willgerodt, F., Ecker, W.: A new approach for generating view generators. In: Design and Verification Conference and Exhibition (DVCon) (2017)
24. Han, Z., et al.: Aspect-oriented design automation with model transformation. In: 2021 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). IEEE (2021)
25. Han, Z., Devarajegowda, K., Neumeier, A., Ecker, W.: IP-coding style variants in a multi-layer generator framework. In: Design and Verification Conference and Exhibition (DVCon) Europe (2020)
26. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical Graph Transformation. J. Comput. Syst. Sci. **64**(2), 249–283 (2002)
27. Bavache, V.B., Han, Z., Hartlieb, H., Kaja, E., Devarajegowda, K., Ecker, W.: Automated SoC hardening with model transformation. In: 2020 17th Biennial Baltic Electronics Conference (BEC), pp. 1–6. IEEE (2020)
28. Dubrova, E.: Fault-Tolerant Design. Springer, Heidelberg (2013). https://doi.org/10.1007/978-1-4614-2113-9
29. Waterman, A., Asanović, K.: The RISC-V instruction set manual. Volume I: Unprivileged v (2020)
30. Devarajegowda, K., Ecker, W., Kunz, W.: How to keep 4-eyes principle in a design and property generation flow. In: MBMV 2019; 22nd Workshop-Methods and Description Languages for Modelling and Verification of Circuits and Systems, pp. 1–6. VDE (2019)
31. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface (2012)