Developing an ESL Design Flow and Integrating Design Space Exploration for Embedded Systems

Falko Guderian and Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Technische Universität Dresden, 01062 Dresden, Germany
e-mail:{falko.guderian, fettweis}@ifn.et.tu-dresden.de

***Abstract*—This paper introduces a systematic development of design flows for embedded systems. The idea of an executable design flow provides a basis for the design automation starting at system level. The aim is to develop, manage and optimize design flows more efficiently. A seamless integration of design space exploration into a design flow is presented coping with the conflicting design goals of embedded systems at electronic system level. It is further shown that an abstract design flow model simplifies a derivation of domain-specific design flows. A novel programming language is introduced allowing for the development of design flows in a visual and textual manner. A case study of the heterogeneous multicluster architecture demonstrates a usage of the design approach and automation. A systematic dimensioning of the multicluster architecture, in terms of the necessary computation resources, is presented in detail. The case study addresses various design problems of future embedded systems at electronic system level. Finally, this paper presents design flow development and design space exploration for embedded systems being systematically, fully integrated, and automated in order to improve a system level design.**

Keywords-electronic design automation, electronic system level, design flow, design space exploration

I INTRODUCTION

It is commonly accepted by all major semiconductor roadmaps that only by raising the design process to higher levels of abstraction will designers be able to cope with the existing design challenges. This leads to an *electronic system level* (ESL) design flow. The term *system level* refers to a use of abstract system functions in order to improve comprehension about a system. Design space exploration (DSE) needs to be integrated in order to trade-off between the conflicting goals of ESL design, such as performance, power consumption, and area [1]. ESL design aims at a seamless transformation of a system specification into a hardware (HW)/ software (SW) implementation [2]. Hence, electronic design automation (EDA) requires a system specification, which is executable in a computer simulation. An *executable specification* is a simulation model of the intended system functions, also called a virtual prototype [3][4].

Today's ESL *design flows*, from now on shortened to flows, are typically based on a specify-explore-refine (SER) methodology [5]. Such flows include a sequence of design steps, from now on shortened to steps, successively refining a system model. Each step solves a design problem, such as application mapping. Moreover, a *specification model* defines the starting point representing the targeted application characteristics and requirements. "Specification model is used by application designers to prove that their algorithms work on a given system platform" [2]. Then, each exploration step creates a design decision continuously increasing the accuracy of the system model. Afterwards, the refined model is passed to the next exploration step. Recently developed EDA environments for ESL design, as proposed in the MULTICUBE project [6] and NASA framework [7], turn away from ad-hoc software infrastructure. The generic EDA systems provide modularization and well-defined interfaces. Despite these advancements, the problem of a large number of possible flow sequences has not been addressed yet.

Since future embedded systems will have an increasing design complexity, the number of steps in a flow is further rising. For example, an optimization of the resource management will require additional steps [8]. Furthermore, the huge design space will draw more attention to an ESL design at an early design stage in order to avoid time-consuming low-level simulations. A systematic methodology to develop, manage and optimize flows promises for a significantly improved design process. In this paper, the approach is denoted as the

design of design flow (DODF). Similar methodologies have been developed in other scientific fields, such as physics [9], mechanical engineering [10], and software engineering [11]. Nevertheless, their degree of automation is limited and the main contribution of this paper is to address this drawback. The aim is to provide an EDA environment increasing the user's productivity.

The remainder of this paper is organized as follows. The related work and design approach are presented in the Sections II and III. In Section IV, the authors introduce the principle of an executable flow. The section also focuses on an explanation of the DODF approach. Then, the introduced concepts are exemplified via a functional exploration of a finite impulse response (FIR) filter. In Section V, the modeling of flows is explained. The idea of abstracting the flows and a corresponding derivation of a domain-specific flow are further introduced. In addition, DSE techniques, applicable to a step and flow, are covered. In Section VI, a visual and textual design flow language (DFL) are presented allowing to develop, manage, and optimize a flow. An according tool flow is introduced afterwards. Finally, Section VII applies the previously developed models and automation tools for an ESL design of the heterogeneous multicluster architecture [12]. The several flows are arranged in a sequence of flows. The flow for the multicluster dimensioning problem will be described in detail.

II RELATED WORK

The related work reviews representative design and specification languages. Moreover, state-of-the-art DSE environments are covered. Then, related studies on meta-modeling are presented. Finally, the use of scripting languages is discussed in the context of EDA.

Specification Languages and DSE Environments

There is a variety of graphical and textual specification languages and frameworks. They can be used to realize ESL design by following a given design methodology. Nevertheless, this is done in a less formal and less generic manner compared to our systematic development of flows. Hence, the reuse and interoperability across tools, designers, and domains are limited. An example is the specification and description language (SDL) [13] allowing for formal and graphical system specification and their implementation. In [14], HW/SW co-design of embedded systems is presented using SDL-based application descriptions and HW-emulating virtual prototypes. Moreover, SystemC [15] and SpecC [16] are system-level design languages (SLDL), which model executable specifications of HW/SW systems at multiple levels of abstraction. These simulation models support SW development. For example, SystemCoDesigner [17] enables an automatic DSE and rapid prototyping of behavioral SystemC models. In [18], a comprehensive design framework for heterogeneous MPSoC is presented. Based on the SpecC language and methodology, it supports an automatic model generation, estimation, and verification enabling rapid DSE. Using an abstract specification of the desired system as starting point, pin-and cycle-accurate system models are automatically created through an iterative refinement at various levels of abstraction. Another example is the specification in a synchronous language, e.g., via Matlab/Simulink. Opposed to that, Ptolemy [19] supports various models of computation to realize executable specifications including synchronous concurrency models. For both examples, DSE has to be realized through a dedicated implementation.

As mentioned in Section I, the MultiCube project [6] and the NASA framework [7] provide a generic infrastructure for ESL design including DSE. Nevertheless, the works do not provide a systematic development of flows and an according design flow language. Hence, they are limited to proprietary flows.

Meta-modeling

Our paper differs to existing work since it is the first using meta-modeling for developing a design flow for embedded systems. Meta-modeling has also been studied to transform from the unified markup language (UML) to SystemC at the meta-model level [20]. This guarantees reuse of models and unifies a definition of the transformation rules. In [21], meta-modeling enables heterogeneous models of computations during modeling. In [22], meta-modeling is used to improve the model semantics and to enable type-checking and inference-based facilities.

Electronic Design Automation

Principally, a general-purpose programming language, such as C/C++, Java, C#, etc., can define a flow via data and control structures. There are different implementation options for a flow description avoiding a unique representation of a flow. Moreover, compilation times prevent from a seamless programming. Hence, a scripting language, tailored to that task, would be rather suited. For example, the major EDA tool vendors Synopsys [23], Cadence [24], and Mentor Graphics [25] provide a scripting language interface for design automation. Therein, the EDA functions are accessible via the language commands in order to build custom flows. The first example is the tool command language (Tcl) [26]. The scripting language has been integrated in the EDA

tools of Synopsys and Mentor Graphics. Tcl is available as open source project without licensing. Another design automation language represents SKILL [27]. SKILL, also a scripting language, has been derived from Lisp, and is integrated in the EDA tools of Cadence. In addition, Perl, Ruby, and Python are used as EDA scripting languages, as presented in [28]. A major drawback of the languages is, they leave it to the designer how to develop, manage and optimize a flow. Hence, the realization of a systematic structure, parallelization, and debugging of flows can differ for each language and designer. This makes the understanding, maintenance and reuse of the flow descriptions a challenging task. This paper addressing the issue by supporting a systematic development of flows via DFL. Furthermore, DSE is directly considered in the language design and implementation, which is not the case for the existing EDA scripting languages.

III DESIGN APPROACH

This section provides an overview of the design approach. It includes two conceptual levels and one instance level related to the terms *method*, *methodology*, and *model*. This is illustrated in the Figure 1. The basic idea is that models and methods are used by a methodology. The classification and relationships will be explained in the following. A composition refers to an element, which is part of another element. Instantiation means that an element is derived from another element. Moreover, the term *meta* is used in order to describe an abstraction of a subject. An example is the meta-data, which means data about data.

The *meta-methodology* defines a methodology realizing another methodology. In Section B, a meta-methodology for the development of flows, also considered as DODF, is introduced. Hence, a flow represents a *methodology*, composed of steps, in order to build the intended design. A *view* allows for a partitioning of a flow resulting in a subset of the steps. Furthermore, a *step* solves a design problem via a method or simulation model. The step consumes inputs and produces outputs. An *input* can be an executable file, configuration, parameter, or constraint. A method or simulation model are compiled into an executable file or callable library. Moreover, an *output* will be a configuration, which is produced when the step has been finished. Each output needs to be validated via a subsequent step including a simulation model or evaluation method. In addition, a control loop between both steps will allow for several design iterations until an output conforms to the pre-defined constraints.

The meta-modeling describes the modeling of the modeling languages. This includes an abstract syntax and the semantics. For example, a *meta-model* enables heterogeneous models of computations in the ESL design, as presented in [29]. In this paper, a meta-model of a flow is introduced in Section A. The intension is to avoid a discussion about the best definition of the term *model*. The considered example is a suitable definition, found in Wikipedia [30]: "A model is a pattern, plan, representation (especially in miniature), or description designed to show the main object or workings of an object, system, or concept.". A flow model is derived from the meta-model. It defines a set of steps and views in order to build a flow. The λ-chart [8], described in Section C, represents a *flow model* following the meta-model. Meta-models can also be defined for the application and architecture models further being implemented in a simulation model and executable specification, respectively. The *application model* represents the functions and the data exchange between the functions of a target application. Moreover, an *architecture model* describes the structure and functions of the intended system, such as the computation architecture, interconnect topology, management infrastructure, communication protocols, etc. Referring to Figure 1, an application and architecture model for future embedded systems are introduced in the Section VII.

A *meta-method* is a method to analyze another method. For example, *meta-optimization* is an optimization method to tune another optimization method. In [31], a genetic programming technique has been used for the meta-optimization in order to fine-tune compiler heuristics. In Section VII, the author applies meta-optimization via an exhaustive search in the *Parameter Tuning* flow in order to find suitable input parameters of a genetic algorithm (GA). Referring to Figure 1, the *method* denotes a technique for solving an ESL design problem. Optimization and estimation methods are used in the case study presented in Section VII.

IV ESL DESIGN FLOW

Early EDA flows were dominated by capturing and simulating incomplete specifications. Later, the logic level and register-transfer level (RTL) synthesis allowed to describe a design only from its behavior and structural representations. However, a system gap between SW and HW design exists since SW designers still provide HW designers with incomplete specifications. An executable specification, such as implemented via C++, SystemC [15], LabVIEW [32], Simulink [33], Esterel [34], Lustre [35], and Rhapsody [36], closed the

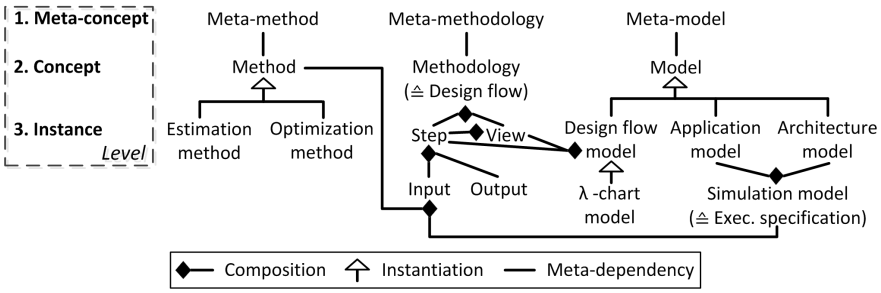

Figure 1. Overview of the design approach.

system gap by describing the system functionality [37]. An ESL flow copes with the design complexity of current multi-processor system-on-chips (MPSoCs). It is expected that the complexity of future many-core SoCs with thousands of cores will further increase the design space [38]. An increasing number of components and their interactions increases the complexity of implementing a many-core SoC flow. The result is a larger number of steps and the inputs/outputs consumed and produced by the steps. In addition, the control structure of a flow will become more complicated. For example, a step can be dependent on multiple steps. Moreover, the variation of multiple parameters/constraints may require nested looping and feedback loops. This section addresses the complexity problem by introducing an executable flow and the DODF approach. The result is a unified methodology to develop, manage, and optimize flows.

A. *Executable Design Flow*

In [1], the authors presented the concept of an integration of DSE into a system-level specification. From that, the idea of an executable flow [39] has been derived. An executable flow denotes a program solving certain design problems and being automatically interpretable by a machine. In an executable flow, methods and simulation models, assigned to steps, are called in the same way instructions of a computer program are called by an interpreter. Predefined methods and models for the steps, e.g., accessible via C++ libraries, would further improve the quality, time and costs of a design. In an executable flow, inputs and outputs are consumed and produced by the steps. The input parameters and constraints control an execution of the steps in a flow. Moreover, an output could comprise a configuration of a refined system model. Since several input values are most likely possible, it results in a huge input or design space of an executable flow. An optimization of the input combinations of each step aims at an adequate step result. Nevertheless, an optimum, comprising all step inputs, is most likely impossible due to the huge design space. This implies several local optima and according design tradeoffs. Moreover, a read access to inputs of a flow will allow for a detection of interfering, inadequate or missing inputs. A further goal is to execute as much as possible steps in parallel. This can be realized for the inputs of a step or by executing independent steps of a flow in parallel. A simple executable flow is illustrated in Figure 2. The flow includes two steps realizing the methods of *Dimensioning* and *Mapping*. First, the dimensioning, implemented, e.g., via an estimation method, extracts an HW architecture from the input configurations of the HW unit options and application. Then, simulation results can be obtained from the mapping of the application onto the HW architecture, as done in the mapping step. Referring to Figure 2, an executable specification implements the system functions necessary to evaluate the system performance.

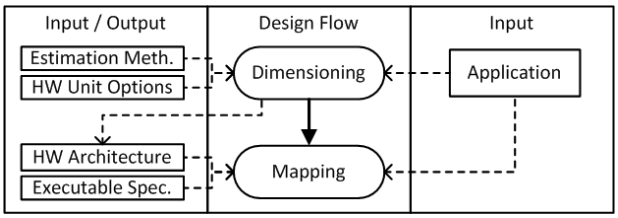

Figure 2. An example of an executable design flow.

B. *Design of Design Flow*

The structure of an executable flow and a methodology for developing flows are incorporated into the DODF approach [39]. The concepts and realizations of DODF are summarized in a hierarchical manner, as seen in Figure 3. The figure shows several members assigned to different hierarchical levels. By moving from the outer part to the inner part of the figure, the concepts are transformed into concrete realizations. The Section C includes an example of a digital filter design illustrating an executable flow and the DODF approach.

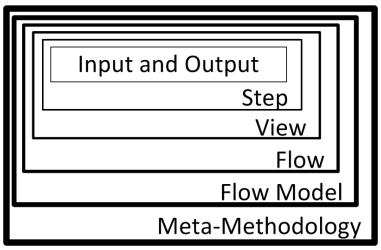

Figure 3. Hierarchy of concepts and realizations in the design of design flow.

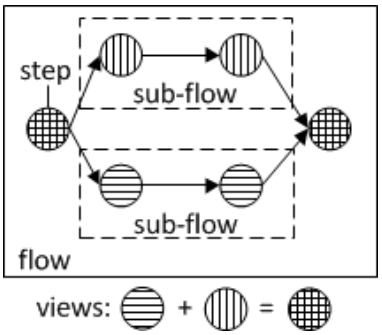

Figure 4. A design flow composed of sub-flows and steps filtered via the views.

First of all, the meta-methodology defines a methodology to create flows. Referring to Section III, the prefix "meta" is used since a methodology is considered as flow. The meta-methodology includes different stages in order to correctly determine and arrange the members defined in the DODF hierarchy. For example, once the steps, their inputs, and their outputs are detected, the steps need to be combined to a flow in order to realize the design goal. In the DODF hierarchy, seen in Figure 3, the flow model is a domain-specific composition of steps and views. The λ-chart [8] is an example of a flow model. As already mentioned before, a model for the modeling of other models is called meta-model. From a meta-model, flow models are created for a specific domain. Then, flows can be derived from the domain-specific flow model. Conceptually, flows are hierarchically composed in order to improve a division of work by assigning a sub-flow or step to specialists in a team. Hence, a flow can be a graph or subgraph with vertexes representing steps. The steps may further represent sub-flows, as indicated in Figure 4. Moreover, each step can belong to a view. Hence, a flow can also include several views, as illustrated in Figure 4. A view represents a level of abstraction in terms of a filter of selected steps. In contrast to a hierarchical division of flows into sub-flow, a view intends extracting a subset of steps assigned to the view. This allows to focus on selective steps and sub-flows. For example, Kogel et al. [40] define the four views: functional view, architects view, programmers view, and verification view. By defining views, a design can be explored from different viewpoints, such as computation topology, interconnect topology, etc. Then, the functionality can be separately analyzed to be explored together in a subsequent design stage. An example is a step assigning the scheduling of computation tasks and load/store tasks to separate views. After the scheduling is explored separately, the results are combined in order to apply the best scheduling technique for all task types.

As mentioned before, a flow is a combination of steps refining a specification model into a targeted system model. Each step uses inputs to apply a method or simulation model, which are compiled into an executable file. An input parameter relates to a description of the structure, behavior, and physical realization of a component or system. Parameters, configuring a design method, are also covered. Furthermore, an input constraint is a restriction of a component or system, such as latency, power consumption, or chip area. Then, the output of a step serves as input for the subsequent step.

As explained before, a flow is derived from a flow model using the meta-methodology and procedure, respectively, illustrated in Figure 5. The idea is to systematically determine, assign, and order sub-flows and the further members of the presented DODF hierarchy, seen in Figure 3. Moreover, an executable flow is built through an algorithmic ordering of the sub-flows and steps. That means, dependencies, loops, branches, etc., realize an execution order of sub-flows and steps in an algorithmic manner. Hence, the ordering of steps realizes a system-level design algorithm based on flow control structures and patterns, respectively, presented in Section B. The meta-methodology glues the members of the DODF hierarchy together in order to systematically follow the DODF approach. Referring to Figure 5, the design goals are first determined and sub-flows are extracted. For example, the design of system components, such as processors, memory, controller, etc., and the design in different levels of abstraction, such as ESL and transaction-level (TL), can be modeled into sub-flows. Then, an algorithmic ordering of the aub-flows needs to be formulated representing the structure of an executable flow. The next stage is to determine the design problems in order to assign each step the corresponding method or simulation model. A method is determined for a step in order to solve a design problem. The simulation mod-

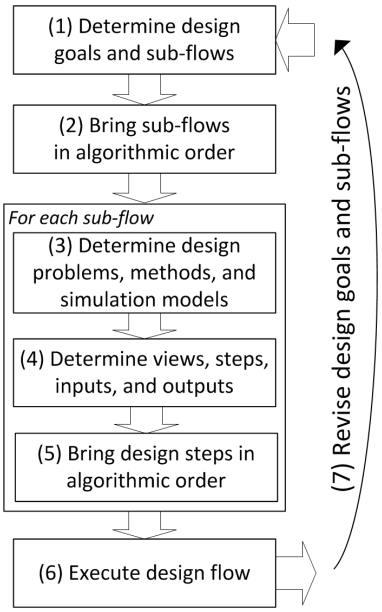

Figure 5. A meta-methodology for the proposed DODF approach.

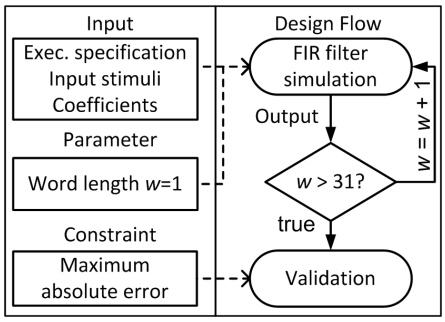

Figure 6. An executable design flow for a functional exploration of the FIR filter.

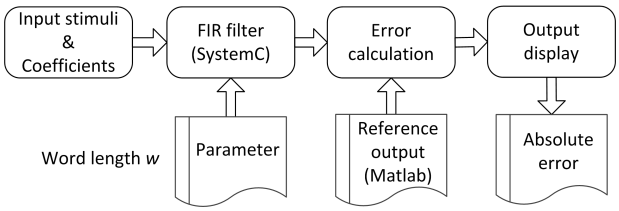

Figure 7. A functional simulation of the FIR filter via SystemC.

els are required for measuring the system performance. Afterwards, each step is assigned a view enabling a horizontal partitioning of the flow. In addition, the inputs and outputs are determined for each step. The next stage finalizes the design of an executable flow by bringing the steps into an algorithmic order. In the end, the flow is executed based on the algorithmic order and variation of the inputs. From the interpretation of the results, the design goals and sub-flows are revised in order to improve the structure and configuration of the flow.

C. A First Example - An FIR Filter

In the following, the flow development is illustrated considering a simple flow. An FIR filter, an ubiquitous digital signal processing algorithm, has been chosen and implemented in a simulation model and executable specification, respectively. Referring to the meta-methodology in Figure 5, the goal and flow are first determined. The goal is to minimize area and power consumption of the memory in an HW implementation of the FIR filter. This is realized via exploring a minimal word length for the bit representation of the FIR filter coefficients. The simple flow is composed of two steps *FIR filter simulation* and *Validation*, as seen in Figure 6. The flow realizes an algorithmic exploration of the FIR filter focusing on the functional view defined in [40]. Hence, the aim is to find the best configuration of the input parameters holding an error constraint. The filter coefficients are provided as real numbers. The word length of each coefficient radix can be varied separately. The step *FIR filter simulation* requires an executable specification, the input stimuli and filter coefficients as inputs. Referring to Figure 6, the step calls an executable specification simulating the FIR function. The simulation performance is evaluated by comparing the output values with a given Matlab reference and calculating a (mean) absolute error, as seen in Figure 7. The output of the step is a mean absolute error representing a degradation compared to the ideal Matlab reference. Referring to Figure 6, the step is executed until the word length w reaches $w = 31$. Then, the *Validation* step finds the best configuration that does not infringe the maximum absolute error constraint. Figure 7 shows the executable specification in terms of a functional simulation of an FIR filter implemented via SystemC [15]. The stimuli represents the input values of the FIR filter. The executable specification is configured with the inputs mentioned before. After the error calculation, a display function returns an absolute error representing the output of the *FIR filter simulation* step.

The following test results are automatically generated by executing the flow. A 16 taps FIR filter with a low-pass characteristic and a cutoff-frequency f_g = 4kHz was configured. In the simulation setup, 1000 uniformly distributed random values are used as input stimuli ranging from 1 to 100. Moreover, the radix of the FIR filter coefficients are jointly varied from 1 to 31 bits.

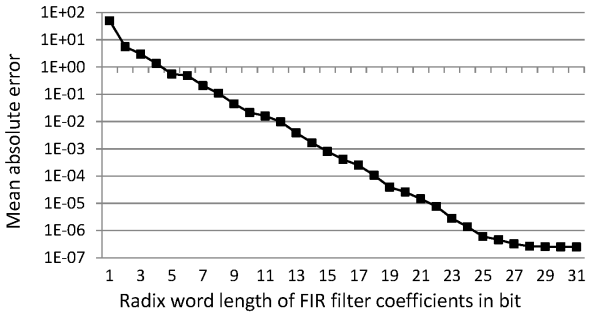

Figure 8. Experimental results of the FIR filter exploration.

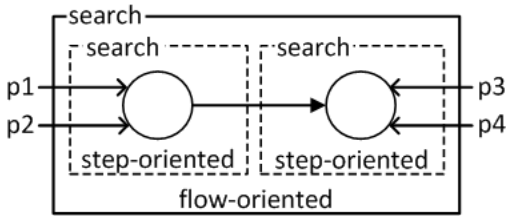

Figure 9. Step-oriented vs. flow-oriented search in an executable flow.

The results are shown in Figure 8. The curve saturates at around 28 bits radix word length with a mean error of $2.6 \cdot 10^{-7}$. In the flow, the maximum absolute error has been set to 10^{-8}. Nevertheless, the parameter variation in the flow needs to have its granularity refined since different coefficients might have different optimal word lengths. A further analysis is presented in the next subsection. The flow is limited to a functional analysis of the FIR filter. Hence, the results should be passed to a flow using executable specifications at a lower level of abstraction, such as TL and RTL.

D. Integrating Design Space Exploration

As mentioned before, an executable flow includes control structures allowing to vary the inputs. Hence, the systematic input variation realizes a design space exploration (DSE). On the one hand, the inputs of a step can be explored limiting the DSE to a step. This refers to a step-oriented search. On the other hand, the aim is to find a suitable combination of all inputs for the steps of a flow. This relates to a flow-oriented search. Step-oriented and flow-oriented search are illustrated in Figure 9. The step-oriented search is limited to the inputs of a step, i.e., the parameters p1-2 or p3-4. Instead, the flow-oriented search aims at exploring all input combinations of a flow, here in the parameters p1-4. The step-oriented search has been focused in this paper. So far, an exhaustive search (ES) and heuristic technique (GA) are developed both applicable to the step-oriented and flow-oriented search. The authors refer to [41] for a comprehensive overview of state-of-the-art search techniques. In general, the DSE methods can be divided into the problem space or the solution/objective space. In the problem space, the parameters, defined in a specification, are considered. An example is a design of a register bank, for which a discrete set of word lengths (columns) and number of words (rows) are available. Now, all possible parameter combinations of columns and rows can be searched within the problem space. In this scenario, the solution space is driven by constraints, such as latency, power, and area. An according DSE strategy can be realized in an unguided or guided manner. ES is a representative of an unguided type allowing for an unbiased view on the design space. Heuristic search, such as hill climbing and GA, is a path-oriented method. It incorporates knowledge in order to guide the search along a path. The advantage is that the intermediate search results may be reused.

As mentioned before, parameters and constraints are similarly represented in a step and flow. Depending on the number of inputs and their range of values, a design space may be divided into sub spaces. The realization of the ES is rather trivial, for example the inputs can be iteratively incremented or taken from a predefined list. In this paper, a GA is presented implementing a heuristic search in the design space. The GA needs to be configured in terms of a minimization or maximization problem. An one-chromosome individual is used to describe the DSE problem. The chromosome includes an one-dimensional array of genes. Each gene denotes an input and the gene value defines an according value. For example, a chromosome $\boldsymbol{g} = (3, 2, 5)$ includes three inputs. The corresponding gene values are in the integer range. Hence, a set or range of values has to be defined for each input. Given a randomly initialized population, the GA generates its offspring via variation. Each chromosome is evaluated by calculating a fitness value. The calculation is done externally in a step and the fitness value is gathered by the GA. In addition, the GA prevents from recalculating already evaluated solutions. Furthermore, variation through an one-point mutation and order crossover enables an iterative improvement of the offspring. In an executable flow, the implementation of a step-oriented and flow-oriented search is realized by an expansion of the executed nodes, namely steps and flows. Figure 10 shows an iterative execution of many steps/flows parallelized via a selection node and

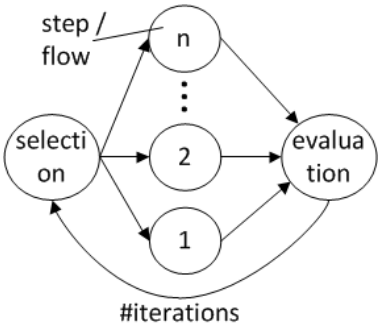

Figure 10. Step-/flow-oriented search via parallelization, synchronization, iteration, and feedback.

synchronized via an evaluation node. In case of an ES, only one iteration is necessary. For each iteration, the GA selects the steps/flows from the population and evaluates the individuals via a provided fitness value. Hence, the selection node performs the genetic operators, such as initialization, mutation, crossover, replacement, and selection. An end of the GA-based search is determined by the number of iterations (generations). This requires a feedback-loop between the selection and evaluation nodes. Further stopping criteria can be included. Moreover, the initialization of the GA population can be used to realize a random (Monte Carlo) search. Hence, the population size corresponds to the number of random samples and the number of generations is set zero.

The step-oriented search is demonstrated via the FIR filter example presented before. The number of taps (#taps) of the FIR filter is #taps=16 and the word length w of each coefficient radix is defined in the range of $1 \leq w \leq 31$ bits. Hence, 31^{16} input combinations motivate for solving the optimization problem via the GA search. Equation (1) defines the fitness (objective) function in terms of a minimization.

$$\gamma \cdot \underbrace{\left(\frac{1}{\#\text{taps}} \sum_{i=1}^{\#\text{taps}} \frac{w_i}{w_{\max}} \right)}_{\text{word length}} + (1-\gamma) \cdot \underbrace{\left(1 - \frac{e_{\text{abs}}^{\min}}{e_{\text{abs}}} \right)}_{\text{absolute error}} \rightarrow \min \tag{1}$$

As mentioned before, Equation (1) needs to be implemented in the FIR filter simulation in order to provide a fitness value for the GA. The fitness function finds a tradeoff between the conflicting goals of a minimal word length of the coefficients and a minimal absolute error. The weight γ realizes a prioritization between both goals. The first term minimizes the word length w_i of the taps i. In the example, the FIR filter requires 16 taps and coefficients, respectively. Then, $w_{\max} = 31$ bits denotes the maximum word length configurable in the FIR filter step. In addition, the second term targets a minimization of the absolute error e_{abs}. Referring to Figure 7, the error is calculated from comparing the filter output in case of quantized coefficients with a non-quantized reference generated via Matlab. Following, $e_{\text{abs}}^{\min}$ represents the minimum absolute error obtained from an FIR filter step by using the maximum word length $w_{\max} = 31$ bits for all coefficients. Figure 11 shows the GA search results in terms of two convergence plots. In the following, the GA is used in order to find a minimum fitness value. The maximum absolute error is set to $e_{\text{abs}}^{\max} = 10^{-3}$ in the FIR filter step. In case the constraint is violated, the FIR filter simulation returns a very large fitness value indicating an invalid solution. In addition, the weight $\gamma = 0.3$ prioritizes the error minimization according to the error constraint introduced before. From the FIR filter results in Figure 8, it is known that an average bit width of $\overline{w} = 15$ reaches a good solution holding the given error constraint. The goal is to reduce the average bit width $\overline{w}$ not violating the constraint. Hence, the bit width of the coefficients is varied in the interval $13 \leq w_i \leq 17$. Furthermore, the GA parameters are set as follows: $pSize = 50$, $nGen = 100$, $mRate = 0.1$, $cRate = 0.8$, and $rRate = 0.5$. In Figure 11, the upper plot shows that the GA converges after 85 generations with a fitness value of 0.6231. Please note, the small decrease of the fitness value at 85 generations is not visible in the figure. From the lower plot in Figure 11, the according absolute error $e_{\text{abs}} = 0.00081$ and average bit width $\overline{w} = 14.3125$ bits can be obtained. Hence, the applied GA search has reduced $\overline{w}$ by almost 5% compared to the result illustrated in Figure 8. In addition, the GA outperforms the average bit width $\overline{w}$, obtained via a Monte Carlo simulation and holding the error constraint, by around 12%. The GA generated 332 different solutions and the DSE finishes after 72 seconds on an Intel Core 2 Duo L7500 with 1.6 GHz utilizing one core. This shows the efficiency of the GA compared to the 5^{16} solutions of an exhaustive search and a solution via Monte Carlo simulation. Nevertheless, an optimal solution can not be guaranteed due to the heuristic nature of a GA.

V MODELING DESIGN FLOWS

This section introduces a meta-model representing an abstract flow model [39]. Moreover, flow patterns are shown in terms of reusable flow structures. Given the meta-model and patterns, a derivation of a flow is illustrated based on the modified λ-chart model [8].

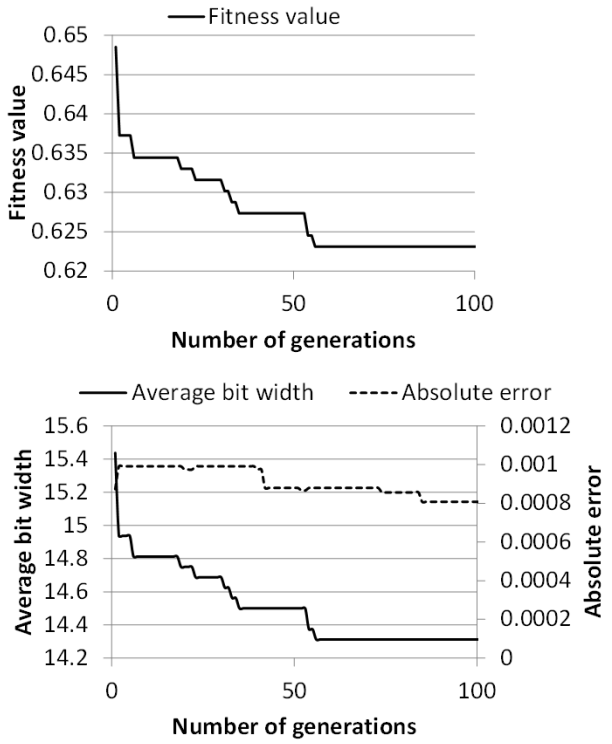

Figure 11. Convergence plots of the GA search in the FIR filter example.

A. *Meta-Model of Design Flows*

A meta-model has been developed in order to provide a minimal set of generic modeling elements necessary to build a flow. The meta-model is described via a UML class diagram, seen in Figure 12. It represents a fundament or kernel of the language design and implementation presented in the Section VI. The language elements relate to the meta classes. The *Element* class contains *Properties* and *Transitions* from/to elements. A transition between two elements is used to model a unidirectional dependency and a property represents an input, output, or further information added to an element. The transition also models a relationship between two flows. Moreover, both *Flow* and *Node* inherit from the element class. The assignment of elements to a view is realized via a property class. Moreover, a flow may include many nodes. Flows may have a nested structure consisting of many flows. This allows to reduce model complexity and to improve the reuse of available flows. Finally, a node represents an executable element, such as step, loop and branch nodes. Loop and branch nodes are further used to describe an algorithmic ordering of flows and steps, as introduced in Section B.

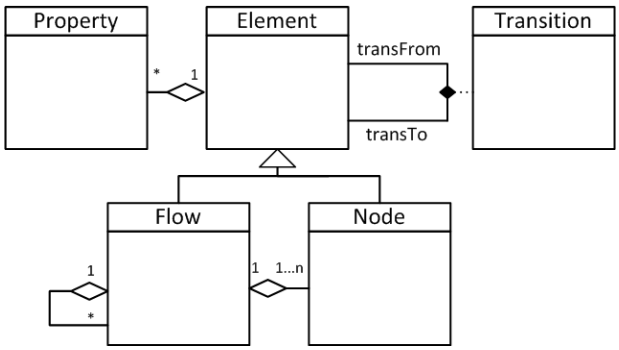

Figure 12. A meta-model for the derivation of design flows.

B. *Design Flow Patterns*

In addition to the meta-model described before, a derivation of recurring structures of flows allows to determine further modeling elements necessary for a systematic construction of flows. The flow patterns, illustrated in Figure 13, are a key enabler of the language design and implementation presented in Section VI. In principle, the patterns describe a parallel, iterative and conditional execution of flows. Pattern (a) models a data dependency between two steps. Hence, the subsequent step is fed with inputs produced by its predecessor. An example is that a scheduling step produces application mappings further being analyzed by a validation step. Moreover, a control dependency models decision making in a flow as seen in pattern (b). It shows a conditional statement deciding for one of two steps depending on the output of a previous step. An example is that only one of the two configurations of a scheduling step will be selected based on the output of a provisioning step. Moreover, pattern (c) describes a divide and conquer approach aiming at a recursive break down of a problem into sub-problems. A possible realization would be that a flow contains several sub-flows representing the sub-problems. In pattern (d), a parallel execution of many steps and the synchronization of the results are described. An example would be to execute the same step with different configurations multiple times in parallel and choosing the best output as input of a subsequent step. Moreover, pattern (e) and pattern (f) consider iterations in a flow. In pattern (e), a step is executed until an end condition reaches. For example, a step increments a parameter in order to find a suitable parameter value. Pattern (f) shows an iterative execution based on a feedback from a subsequent step. The information may allow for changing the selected inputs in order to improve a step result.

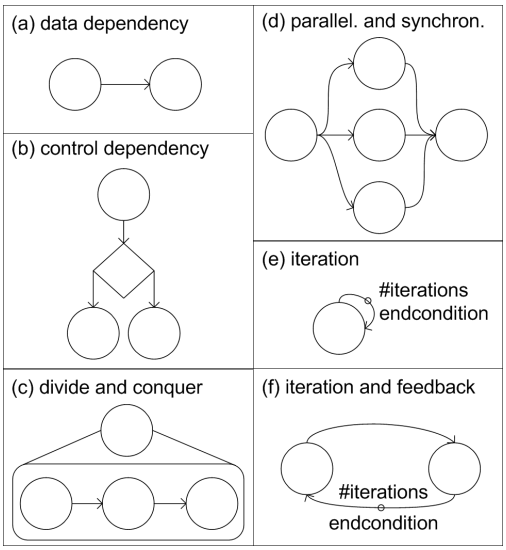

Figure 13. Reoccurring structures (patterns) in design flows.

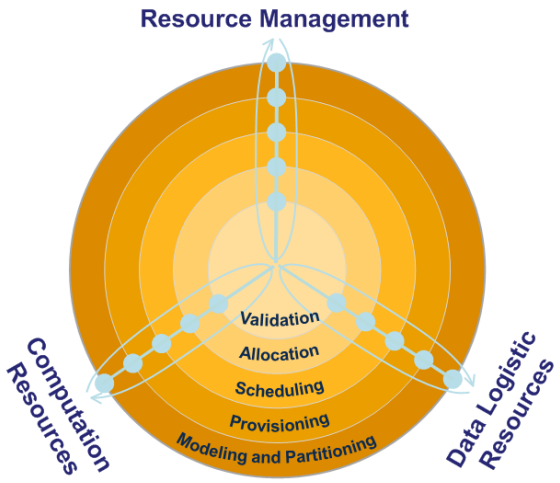

Figure 14. The modified λ-chart [8] - A model of design abstraction and exploration.

C. Domain-Specific Design Flow

In a previous work [8], the authors introduced the *λ-chart*, which represents a model of design abstraction and exploration. It addresses an ESL design of MPSoCs and future many-core SoCs at an early stage. The motivation was to provide the designer with a flow model allowing for a clear definition of the steps and a separation of the important system functions. Therefore, an *administration* view was included in order to highlight the rising importance of management functions in embedded systems. The model further allows to combine the different steps of a flow. In the following, the λ-chart has been slightly modified in order to focus more on the design and exploration of the system resources, as illustrated in Figure 14. In addition, the term administration has been replaced by a more management-centric point of view. Hence, the λ-chart defines three views allowing to separate the orthogonal system functions. A *resource management* view considers tasks for planning, assignment, monitoring, and control. Instead, a *computation resources* view relates to the code execution. Moreover, a *data logistic resources* view addresses a design of data storage and data exchange between components. Furthermore, the concentric bands underline the five steps of a unified process. The *modeling and partitioning* step describes a starting point in order to build the representations of the system structure and behavior. *Partitioning* focuses on the parallelization of applications. Following, *provisioning* means to select the type and number of components and behavior necessary to fulfill the purpose of the intended system. In *scheduling*, a temporal planning of the computation, data logistics and management is applied. This includes both the application and architectural components, such as determining an execution sequence, power-aware planning, monitoring, etc. Moreover, the *allocation* step focuses on spatial planning, such as placement and packaging of components, and application binding. Finally, *validation* proves whether the system fulfills a previously defined purpose. The authors refer to [8] for a more detailed explanation.

The λ-chart follows the meta-model presented in Section A. That means, a step is derived from the node element and a flow is a sequence of steps connected via transitions. Moreover, a view is modeled via the property element. An example of a flow, depicted in Figure 15, demonstrates the derivation of a flow from the λ-chart. Three steps, limited to the *computation resources* view, have been chosen. The combination of the steps and a connection via transitions build the flow. The block diagram in Figure 15 shows an equivalent representation of the flow. In addition, control primitives, such as a branch node (if-then-else, switch-case) and loop node (for/while), are inserted in a flow enabling a parallel, iterative and conditional execution of the flow. This allows to realize the flow patterns presented before. In Section IV, the DODF approach was introduced, giving the designer a methodology to select appropriate flows, views, steps, etc. The control structure is build via an algorithmic order of the steps. Figure 16 details the instantiation from the *Element*, *Transition* and *Property* classes defined in the meta-model. Figure 16 (left) shows

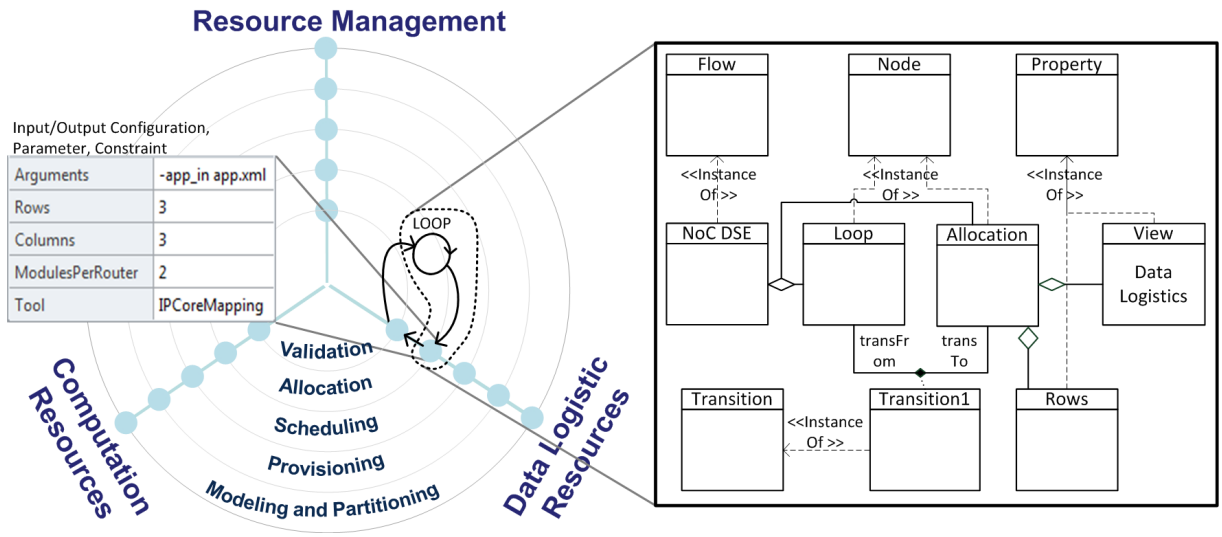

Figure 16. An example of a λ-chart flow with instantiation from the meta-model.

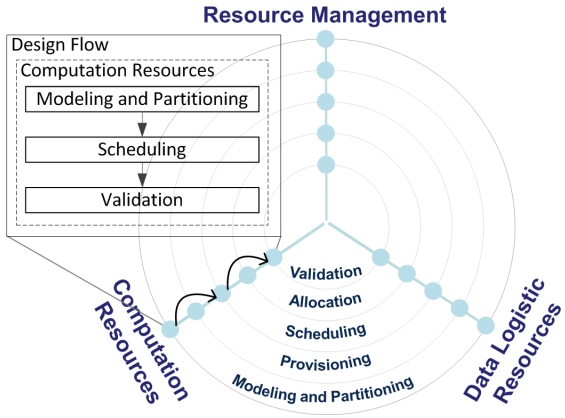

Figure 15. An example for the derivation of a flow in the modified λ-chart.

a flow traversing the *allocation* and *validation* steps iteratively. The DSE is restricted to the *data logistic resources* view. In the following, a limited part of the flow, marked by a dotted line, is considered. Referring to Figure 16 (right), the example focuses on the allocation step, loop node, and transition from the loop to allocation. The loop node controls an iteration of the input parameters of allocation and includes an exit condition. Moreover, the flow is named network-on-chip (NoC) DSE. NoC is a promising network design approach for scaling from MPSoC to many-core systems because the efficient communication infrastructure supports a large amount of IP cores [42, 43]. As mentioned before, an assignment of the allocation step to a view is realized via the *Property* class. The step also includes properties, such as the number of rows in a NoC. Hence, the properties are used as input parameters of a step.

VI ESL DESIGN AUTOMATION

A comprehensive list of academic and commercial EDA environments for ESL design can be found in [2]. Modern environments address DSE but with the limitation to a proprietary implementation for a specific design problem, such as optimization of the application mapping. Recent research introduces generic infrastructures turning away from ad-hoc software [6, 7]. Nevertheless, the complexity of flows for future embedded systems is not yet considered. The large number of flows, steps, inputs and outputs requires a more systematic development. In addition, commercial EDA systems allow for a flexible and efficient implementation of flows via scripting languages. The major drawback of academic and commercial EDA systems is that no systematic development, management and optimization of flows is supported. The user is either dependent on a proprietary implementation or has to develop a representation of a flow by oneself. This paper presents two programming languages [39] addressing these problems and supporting all aspects of our DODF approach. Therefore, the user is supported in developing, managing, and optimizing a flow. This includes flexible and efficient realization of DSE strategy in the flow via little program code. First of all, a visual programming language is introduced. This language has been evolved to a textual programming language, called

design flow language (DFL). A tool flow, enabling DFL, is presented afterwards.

A. Visual Programming of Flows

A visual programming of flows has been implemented via a graphical prototype based on Microsoft Visio by the authors [1]. It realizes the concepts introduced in the Sections IV and V. The implementation allows to instantiate steps and flows via drag-and-drop and copy functions using the λ-chart model. The graphical user interface (GUI) corresponds to the visualization in Figure 16 (left). The construction of a flow from the GUI has been realized via the visual basic for applications (VBA) programming language by detecting the dependencies between the steps and reading the properties of the steps. The prototype includes an import/export function in order to load and store the flows based on a predefined XML-format. The definition of the XML-format is explained via a simple flow, illustrated in Listing 1. The flow corresponds to the Figure 16 (left). The XML-file is read by an interpreter program implemented in C++. The interpreter allows for a sequential and parallel execution of the steps. Referring to Listing 1, the *flow* and *node* tags follow the meta-model presented in Section A. The step and loop nodes are connected via transitions and include many properties. Moreover, the loop node requires a loop/exit body and an exit condition in order to traverse the flow iteratively. In a *property* value, expressions and system functions are used to read and modify variables, directories, and files during a step execution.

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <flow>
3    <node name="My Allocation">
       <property name="Step" value="Allocation"/>
5      <property name="View" value="Data Logistic
           Resources"/>
       <property name="Rows" value="3"/>
7      <property name="Arguments" value="-app_in
           lambda\\apps_state.xml ..."/>
       <property name="Tool" value="IPCoreMapping
           "/>
9      <!-- ... -->
     </node>
11   <node name="My Validation">
       <property name="Step" value="Scheduling"/>
13     <property name="View" value="Data Logistic
           Resources"/>
       <!-- ... -->
15   </node>
     <node name="My Loop">
17     <property name="type" value="LOOP"/>
       <property name="loop_body" value="Rows=
           Rows+1; ..."/>
19     <property name="exit_condition" value="
           Rows==4; ..."/>
       <property name="exit_body" value="
           renameDir(lambda\\maps,Rows); ..."/>
21   </node>
   <transition source="My Allocation" target="My
       Validation"/>
23 <transition source="My Validation" target="My
       Loop"/>
   <transition source="My Loop" target="My
       Allocation"/>
25 </flow>
```

Listing 1. XML source code imported/exported by the visual programming prototype.

Referring to Listing 1, the step "My Allocation" (lines 3-10) and the step "My Validation" (lines 11-14) are created. Therein, several properties are defined, such as *Step*, *View*, etc. Moreover, the *Rows* property (line 6) is initialized to three. Together with the Arguments (line 7), Rows will be used as input of the *IPCoreMapping* tool (line 8). Moreover, the loop node (lines 16-21) defines several expressions in order to increment the *Rows* property (line 18), to check for the exit condition (line 19), and to define an action after the exit (line 20). Finally, the flow is constructed by connecting the steps via transitions (lines 22-24).

Nevertheless, the XML-format makes it inconvenient to program multiple expressions, nested conditions, nested loops, and feedback loops. In addition, a reuse of flows and steps is not supported. The limitations motivated for an evolution towards the DFL representing an efficient and flexible programming language.

B. Design Flow Language (DFL)

DFL is specially targeted to a development, management and optimization of flows including the necessary control and automation capabilities. Moreover, design space exploration (DSE) is directly considered in the language design and implementation. The requirements and structure of DFL are shortly introduced in the following. A simple flow example illustrates the use of the language. For more details on the language, the authors refer to [39].

Language Requirements

The purpose of DFL is to make the design of future embedded systems more flexibly and efficiently via a systematic development of flows. This includes management and optimization capabilities. The requirements are summarized in the following. A clean syntax increases the user's productivity. Program commands for the construction of flows are necessary. As in modern programming languages, control structure and program modularization enable more complex applications. Moreover, an acceleration of flows via parallelization

should be realized. The use of DSE techniques within a step or flow will allow to find an optimal or feasible solution in design spaces with different complexity. Further requirements relate to the EDA tools and design data accessible via DFL. An executable file or library needs to be assigned to a step. Moreover, design data should be accessible via data structures, files, and data base operations. In addition, some kind of inter-process communication serves as interface between the EDA tools. Finally, non-functional requirements address an access from/to other programming languages. Moreover, data analysis and debugging support will be beneficial in a flow development.

Language Structure

The DFL is an imperative (procedural) programming language read by an interpreter program. The interpreter controls an execution of the steps defined in a flow. The syntax is derived from the C/C++ programming language widely known in HW/SW programming. The *Flow*, *Step*, *Property* and *Transition* classes, defined in the meta-model and introduced in Section A, have been integrated in the language design and implementation. Modularization is realized via subroutines and an #include statement. Basic data types (bool, int, double, string) and complex data types (vector, Flow, Step) are available. DFL is further a structural programming language supporting a full set of control primitives, such as for, while, if-then-else and switch-case. The language includes a limited number of keywords and various input/output names are reserved for the step and flow. DFL additionally supports typical arithmetic operators, logical operators, and vector indexing. Moreover, commands are case sensitive and single statements must be ended with a semicolon.

A Simple Design Flow in DFL

In the Listing 2, a simple flow is described in DFL illustrating its structure. The program accomplishes an execution of two dependent steps in a flow, which corresponds to Figure 16 (left). Lines 2-8 relate to the configuration of an *allocation* step. This includes an assignment of an executable file, called alloc(.exe), to the step (line 3). The executable requires arguments (line 6) and an input (line 7) in order to solve the IP core mapping problem (line 4). In addition, the *View* parameter corresponds to the λ-chart in Figure 16. Since the step allows for several input combinations, here indicated via the *rows* vector (line 7), it is configured for a parallel execution (lines 10-13). A *space* vector contains the variables defining the input combinations (lines 10-11). The input parameter *HPCJob* (line 13) configures an available high performance cluster (HPC) environment for a parallel execution of the steps. Then, a *validation* step (lines 15-16) is instantiated. Further assignments to the step are left out for simplification. Finally, the flow is constructed (lines 19-22) and executed (line 24). The steps need to be added to the flow (line 20) and the execution order is determined via the *connect* function (line 21). Line 22 saves the flow description in the visualization of compiler graph (VCG) format [44] allowing to check the flow structure.

```
1  /******** ALLOCATION STEP ********/
   Step s1 = Step("Allocation");
3  s1.add("Execution","alloc");
   s1.add("Tool","IPCoreMapping");
5  s1.add("View","Data Logistic Resources");
   s1.add("Arguments","-app_in lambda\\apps_state
       .xml ...");
7  vector<int> rows = [3:4];
   // ...
9  /******** PARALLEL EXECUTION ********/
   vector<string> space;
11 space.push_back("rows");
   s1.add("Space","space");
13 s1.add("HPCJob", "true");
   /******** VALIDATION STEP ********/
15 Step s2 = Step("Validation");
   s2.add("Execution","valid");
17 // ...
   /******** FLOW CONSTRUCTION ********/
19 Flow f;
   f.add(s1); f.add(s2);
21 connect(s1,s2);
   f.save("vcg", "flow.vcg");
23 /******** FLOW EXECUTION ********/
   execute(f);
```

Listing 2. Simple design flow in DFL.

DFL Tool Flow

In the following, the tool flow for the DFL is presented. As typical for modern programming languages, it is separated into frontend, middle-end, and backend. Figure 17 illustrates the tool flow. The frontend includes a scanner and parser to validate the DFL syntax. The scanner splits the DFL source code into tokens by recognizing lexical patterns in the text. GNU Flex [45] has been used to generate the scanner (lexical analyzer). Then, the parser applies syntax-rule matching. The parser has been generated using GNU Bison [46]. From the parsing results, an abstract syntax tree and a statement list are derived. In addition, a symbol table holds information about the program. The statement list and symbol table allow to interpret and optimize the program code,

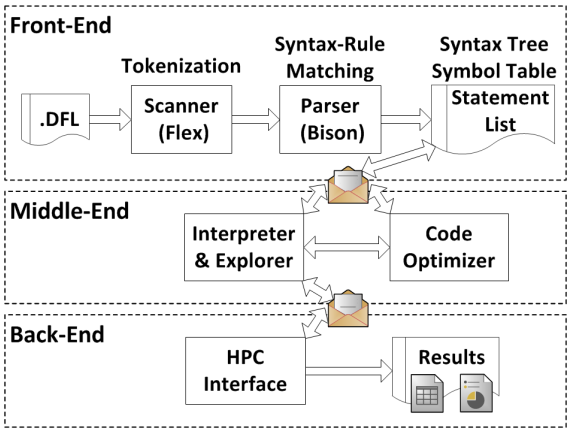

Figure 17. Tool flow for the design flow language.

as done in the middle-end. The interpreter is responsible for type checking, type erasure (conversion), and expression evaluation. The code optimization refers to an exploitation of the step-level and flow-level parallelism. As mentioned before, the interpreter supports an export of the flow structure in the VCG format [44] in order to visualize the graph. Moreover, the explorer includes an exhaustive, random and heuristic search allowing to explore design spaces with different complexity. Finally, the backend provides functionality executing a DFL program on a single computer or HPC. After a step execution, according design and validation data will be available for a further analysis. The next stage is to merge DFL and model/method design into an integrated development environment (IDE), presented in [39]. Therein, the design methods and simulation models are implemented via a native language, such as C/C++, in order to fulfil the critical performance requirements. The aim is to compile an executable or library and assign it directly to a DFL step in one IDE realizing a seamless development. Then, the flow can be executed, tested, and optimized in the IDE. The DFL implementation includes a full set of language features. Open topics relate to the implementation of performance analysis functions, plotting functions, and database access. Furthermore, a future DFL revision needs to address name spacing avoiding naming conflicts.

VII DESIGN FLOW CASE STUDY

This section demonstrates the concept of an executable flow and the DODF approach under realistic conditions. The case study targets an ESL design of the heterogeneous multicluster architecture, as introduced by the authors in [1]. The multicluster architecture represents a promising candidate for future embedded many-core SoCs [12]. The outline of this section is as follows: First, a description of the application and architecture model forms the basis of the underlying simulation models. Next, an according sequence of flows is introduced showing a separation of the addressed design problems. This allows to solve the complex problems more flexibly and more efficiently as compared to a proprietary and fully integrated design flow. Due to a lack of space, only the dimensioning of the multicluster architecture is selected for a more detailed explanation in terms of a design methodology, flow description, and the experimental results.

A. *Application and Architecture Model*

The models consider functionalities of the three views defined in the modified λ-chart, seen in Figure 14. The application model includes multiple, concurrently running applications and threads, respectively. A thread is represented by a high-level task graph and it sequentially executes tasks. Threads are only synchronized before or after execution. Then, a task is an atomic kernel exclusively executing on an intellectual property (IP) core, e.g., processing element (PE), memory (MEM) interface, control processor (CP) interface, etc. Tasks produce and consume chunks of data accessed via shared memory. Side effects are excluded by preventing access to external data during computation.

As shown in Figure 18, the architecture model is a heterogeneous set of multiprocessor system-on-chips (MPSoCs) and clusters, respectively. The management unit (MU) represents an application processor and includes a load balancer aiming at equally distributing thread load amongst the clusters. Moreover, an MPSoC contains heterogeneous types and numbers of IP cores. In the model, each MPSoC contains a network-on-chip (NoC) connecting the IP cores. Moreover, each cluster includes a CP responsible for dynamically scheduling arriving tasks to the available IP cores. The CPs are directly connected to the MU. The heterogeneous multicluster architecture, seen in Figure 18, includes a regular 2D mesh NoC. Each tile contains a router and n modules (IP cores). A module can be an MEM, CP, or PE, such as general purpose processor (GPP), digital signal processor (DSP), application-specific integrated circuits (ASIC), etc.

B. *Sequence of Design Flows*

This case study is composed of five flows using different design methods and system models. Figure 19 illustrates a sequence of the flows. Further flows can be added, such a memory optimization. The heteroge-

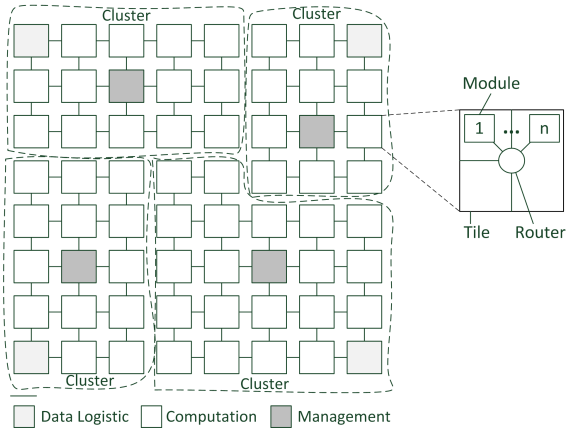

Figure 18. An architecture model for the heterogeneous multicluster.

neous multicluster architecture implies a wide diversity in terms of structural, behavioral (functional) and physical parameters. DFL programs have been developed for the flows. Therein, the view and step definitions follow the modified λ-chart model. Referring to Figure 19, this case study addresses input parameters of the design method, structural design, behavioral design, and physical design. In the following, the flows are shortly introduced and a DFL program for the sequence of flows is presented. The rest of this section will focus on the multicluster dimensioning.

- *Parameter Tuning* aims at finding the best tool parameters for a GA solving the IP core mapping problem [47];
- *Multicluster Dimensioning* creates a heterogeneous multicluster architecture by distributing the anticipated application load among clusters and solving the optimization problem via a genetic algorithm (GA) and mixed-integer linear programming (MILP) formulation [48];
- *IP Core Mapping* places IP cores in an 1-ary n-mesh NoC constrained by the number of modules at each router. The optimization problem is solved via a GA and MILP formulation [47];
- *NoC Arbitration* and *Multicluster Load Balancing* aim at finding suitable behavioral schemes from a selection based on simulation results. *NoC Arbitration* compares a locally fair with a globally fair arbitration scheme [49]. In addition, flit-based and packet-based switching are considered. *Multicluster Load Balancing* compares different estimators of cluster load, such as response time and queue size, used in the load balancing scheme of an MU.

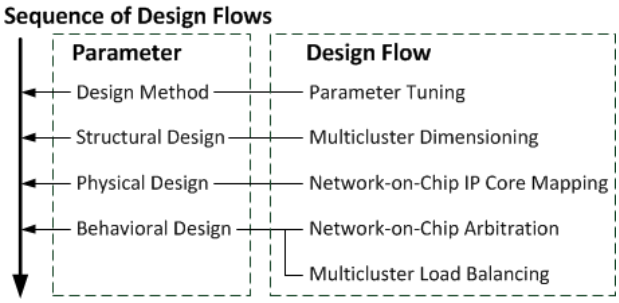

Figure 19. The sequence of flows in the case study.

In the following, a DFL program for the sequence of the flows is presented. The flows *Parameter Tuning* and *Multicluster Load Balancing* are used as examples. The source code of *parameter_tuning.dfl*, shown in Listing 4 in the appendix, gives a deep insight into a flow developed in DFL. Referring to Listing 3 (lines 2-3), the #include directive allows to insert predefined DFL source code as mentioned in the previous section. The variables *tun* and *bal* are declared in one include file and they represent the predefined flows of the *Parameter Tuning* and *Multicluster Load Balancing*. After the #include, an execution sequence is scheduled by inserting an identifier for each flow in the vector *flow_order* (lines 5-9). Thereafter, the vector is iterated (lines 11-42) and a switch-case statement (lines 17-38) lists the available flow choices. If a flow matches a case statement, it is executed and a status message is displayed (lines 40-41). The example further includes two specific inputs and vectors, respectively (line 15). The elements of each vector are used for a DSE purpose, such as searching for the best configuration. The DSE is declared in the steps. The input *arch_in* represents a set of available architecture configurations. The elements in the vector are used as parameter values for the *tun* step (line 21) and the *bal* step (line 28). In addition, the vector *config* includes different configurations of the simulation setup for the *bal* step in order to select a suitable load balancing scheme (line 28). The sequence of flows can be further extended in terms of additional flows, inputs, and, commands.

C. Multicluster Dimensioning

Given a set of target applications, the *Multicluster Dimensioning* flow realizes a provisioning of resources in the heterogeneous multicluster architecture [48]. The aim is to generate an appropriate distribution of the applications onto the clusters containing different types and numbers of PEs. The E3S Benchmark Suite [50] is

used as basis of the applied application scenario. E3S is largely based on data from the Embedded Microprocessor Benchmark Consortium [51]. The included task graphs describe periodic applications. The 20 applications range from automotive, industrial, telecommunication, networking to general-purpose applications. An application scenario is built from the concurrently running task graphs.

An overview of the methodology is illustrated in Figure 20. Besides optimization of the multicluster architecture, the flow applies further methods, such as estimation, (architecture) refinement, simulation, and validation. Referring to Figure 20, the first step is to extract a parallelism value matrix $\mathbf{\Phi}$ via parallelism analysis, introduced by the authors in [48]. The matrix is used as input for the optimization via a GA and MILP formulation. Given the optimized cluster configurations, the selected IP cores are used to generate an multicluster architecture. Then, the dynamic mapping of an application onto the refined architecture is simulated. Each task of an application is dynamically mapped onto an IP core at runtime assuming a point-to-point communication protocol between the directly connected IP cores. Each task is executable on at least one IP core of the refined architecture ensuring schedulability. Moreover, a task execution is prioritized based on its deadline. Afterwards, the mapping results are validated by an average thread response time quantifying the system performance. Response time defines the time from the request of a thread until its end including a possible network delay.

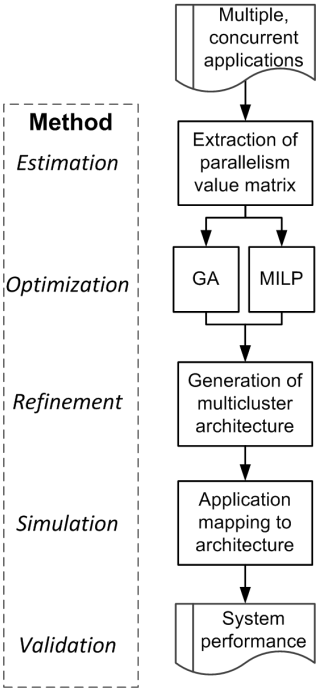

Figure 20. Methodology of the *Multicluster Dimensioning* flow.

A compact flow description, seen in Figure 21, is realized via the modified λ-chart. The flow focuses on a suitable computation infrastructure for the heterogeneous multicluster architecture. Hence, DSE is limited to the *computation resources* view. The *modeling and partitioning* step serves as a starting point without any further purpose. In the *provisioning* step, a target application and the available IP cores are used to generate the heterogeneous multicluster architecture. As mentioned before, the optimization problem is solved via a GA and MILP formulation. The subsequent *scheduling* step performs an application mapping via simulation. An according simulation model performs both a temporal and spatial mapping of the tasks to the available PEs dynamically at runtime. The results are analyzed in the *validation* step. Referring to Figure 21, a loop node increments a maximum allowed number of PEs in a cluster ($\#PEs_{max}$). For the simulations, the value range of the input constraint is set to $3 \leq \#PEs_{max} \leq 7$.

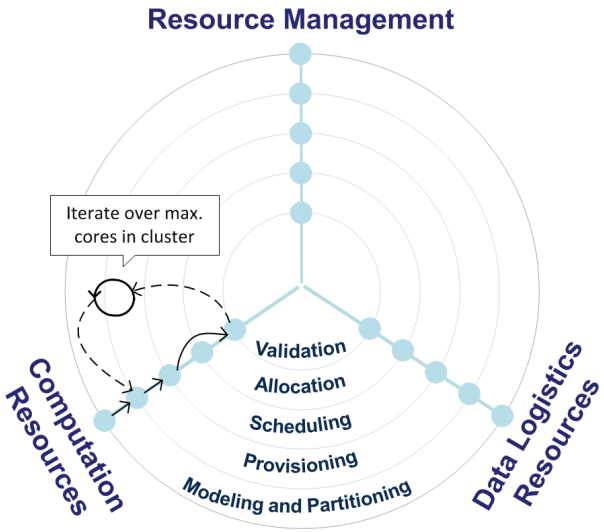

Figure 21. Overview of the *Multicluster Dimensioning* flow via the modified λ-chart.

In the literature, to the best knowledge of the authors, multicluster dimensioning was not yet applied for the E3S Benchmark Suite [50]. In order to compare the re-

sults, a single-cluster configuration with nine PEs is provided as reference in Figure 22. It has also been generated with the *Multicluster Dimensioning* flow. Application mapping onto the single-cluster architecture results in over 40 % thread cancelation. Then, using the thread response time as a metric would be meaningless, hence the total amount of PEs is considered as a reference.

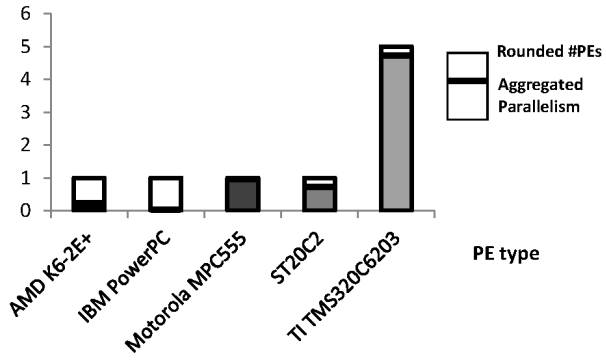

Figure 22. Single-cluster reference for the *Multicluster Dimensioning* flow.

Figure 23 shows the validation results in terms of a total number of clusters/PEs and (average) thread response time. The latter includes the impact of the dynamic scheduling scheme. The GA has been used to solve the multicluster dimensioning problem. All values have been normalized to the largest occurring value. The selection of a suitable solution bases on a tradeoff between the conflicting goals of a minimum number of resources and a minimum thread response time. In the figure, $\#PEs_{max} = 7$ (red arrow) is selected as the best tradeoff. Its application mappings did not produce aborted threads. It includes a minimum number of clusters of three and PEs of eleven. As mentioned before, each cluster contains a CP further increasing the number of resources in the system. In the result, the number of clusters and PEs do not change for the larger $\#PEs_{max}$ values. But due to its heuristic nature, the GA produced the best solution in terms of a thread response time for $\#PEs_{max} = 7$. The resulting configuration, depicted in Figure 24, represents a heterogeneous multicluster solution since all clusters are heterogeneous in terms of PE types (depicted by different shades of grey). In the Figure 24, it is shown that the PEs of the PE types *AMD K6-2E+* and *IBM PowerPC* are marginally used. The both GPPs are able to execute most of the tasks in the benchmark. The remaining PEs are well utilized using the anticipated application load based on the average parallelism values. The configuration shows improvement potential in the cluster C2. A solution would be to exclude the *IBM PowerPC* from the mapping option table in order to reduce the number of PEs in the cluster by one PE. This requires that the PE can be replaced and no additional PE is necessary to perform the tasks assigned to the *IBM PowerPC*. Hence, the total number of PEs decreases to ten.

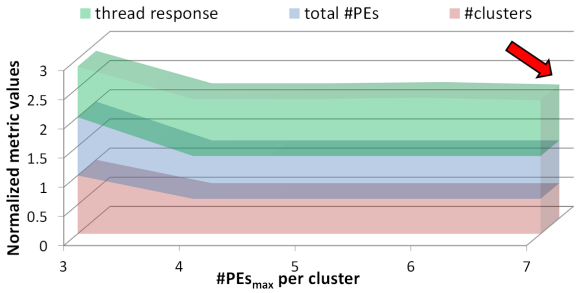

Figure 23. Normalized results of the *Multicluster Dimensioning* flow.

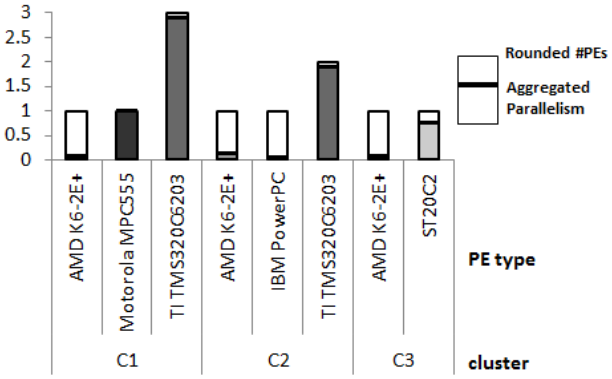

Figure 24. Best multicluster configuration of the *Multicluster Dimensioning* flow.

VIII CONCLUSION AND OPEN TOPICS

The large number of inputs and steps in the flows for future embedded systems necessitates the development of a systematic design of design flow (DODF) approach. Then, the concept of an executable flow allows for executing steps in the same way instructions of a program are processed. Both contributions of this paper are exemplified via a functional exploration of an FIR filter. Afterwards, the modeling principles of flows are explained. The idea of abstracting the flows and a corresponding derivation of a domain-specific flow are focused. The concepts are the motivation for a visual and textual design flow language. The design automation allows for a development, management, and optimization of flows.

Design space exploration is directly considered in the language design and implementation. Finally, a case study demonstrates a realistic ESL design of the heterogeneous multicluster architecture. The five flows are arranged in a sequence of flows. Each flow outputs experimental results representing suitable solutions for the individual design problems.

In the rest of this paper, a discussion outlines the future work. An open topic relates to the further development of DFL towards additional language features, such as name spacing, profiling, etc., allowing for more complex applications. In addition, the language should provide advanced access and functions to analyze the design data. It would be beneficial to support more DSE techniques, such as simulated annealing, hill climbing, etc. In addition, the flow-based search is an open topic. The implementation of DFL comprises a full set of language features opposed to the visual language, which requires several adjustments, such a support of sub-flows in a flow. In future, the design flow development should be extended towards a high-level synthesis for embedded systems.

VIII APPENDIX: DFL FLOW EXAMPLE

The appendix illustrates the DFL source code for the *Parameter Tuning* flow through Listing 4.

REFERENCES

[1] F. Guderian and G. Fettweis, "Integration of design space exploration into system-level specification exemplified in the domain of embedded system design," in *Proceedings of International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, Aug. 2012.

[2] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.

[3] R. Ernst, "Automatisierter entwurf eingebetteter systeme," *at - Automatisierungstechnik*, pp. 285–294, jul 1999.

[4] B. Bailey, G. Martin, and A. Piziali, *ESL design and verification: a prescription for electronic system-level methodology*, 1st ed., W. Wolf, Ed. Morgan Kaufmann, 2007.

[5] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*. Prentice-Hall, Inc., 1994.

[6] W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, M. Wouters, C. Kavka, L. Onesti, A. Turco, U. Bondi, G. Marianik, H. Posadas, E. Villar, C. Wu, F. Dongrui, Z. Hao, and T. Shibin, "Multicube: Multi-objective design space exploration of multi-core architectures," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2010, pp. 488 –493.

[7] Z. J. Jia, A. Pimentel, M. Thompson, T. Bautista, and A. Nunez, "Nasa: A generic infrastructure for system-level mp-soc design space exploration," in *Proceedings of Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Oct 2010, pp. 41 –50.

[8] F. Guderian and G. Fettweis, "The lambda chart: A model of design abstraction and exploration at system-level," in *Proceedings of International Conference on Advances in System Simulation (SIMUL)*, 2011, pp. 7–12.

[9] R. A. Fisher, *The Design of Experiments*. Oliver and Boyd Ltd., Edinburgh, 1935.

[10] G. L. Glegg, *The Design of Design*, 1st ed. Cambridge University Press, 1969.

[11] F. Brooks, *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley, 2010.

[12] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The multicluster architecture: reducing cycle time through partitioning," in *IEEE/ACM International Symposium on Microarchitecture (Micro)*, 1997, pp. 149–159.

[13] ITU-T, *Recommendation Z.100 (08/02) Specification and Description Language (SDL)*, International Telecommunication Union (2002).

[14] S. Traboulsi, F. Bruns, A. Showk, D. Szczesny, S. Hessel, E. Gonzalez, and A. Bilgic, "Sdl/virtual prototype co-design for rapid architectural exploration of a mobile phone platform," in *Proceedings of international SDL conference on design for motes and mobiles*, 2009, pp. 239–255.

[15] A. S. Initiative. (26 May 2013) Systemc, osci. [Online]. Available: http://www.systemc.org/

[16] D. D. Gajski, R. Zhu, J. Dömer, A. Gerstlauer, and S. Zhao, *SpecC Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[17] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models," in *Proceedings of the 45th annual Design Automation Conference*, ser. Proceedings of Design Automation Conference (DAC), 2008, pp. 580–585.

[18] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-chip environment: a specc-based framework for heterogeneous mpsoc design," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 5:1–5:13, Jan. 2008.

[19] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, jan 2003.

[20] L. Bonde, C. Dumoulin, and J.-L. Dekeyser, "Metamodels and mda transformations for embedded systems." in *FDL*, 2004, pp. 240–252.

[21] D. Mathaikutty, H. Patel, S. Shukla, and A. Jantsch, "Ewd: A metamodeling driven customizable multi-moc system modeling framework," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 33:1–33:43, May 2008.

[22] D. Mathaikutty and S. Shukla, "Mcf: A metamodeling-based component composition framework–composing systemc ips for executable system models," *IEEE Transactions on VLSI Systems*, vol. 16, no. 7, pp. 792 –805, july 2008.

[23] "Synopsys Inc." 26 May 2013. [Online]. Available: http://www.synopsys.com

[24] "Cadence Design Systems Inc." 26 May 2013. [Online]. Available: http://www.cadence.com/

[25] "Mentor Graphics Inc." 26 May 2013. [Online]. Available: http://www.mentor.com/

[26] B. Welch, *Practical Programming in Tcl and Tk*, 4th ed. Prentice Hall, 2003.

[27] T. Barnes, "Skill: a cad system extension language," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, jun 1990, pp. 266 –271.

[28] Q. Nguyen, *CAD Scripting Languages: A collection of Perl, Ruby,Python,TCL & SKILL scripts*. Ramacad Inc.

[29] A. Sangiovanni-Vincentelli, G. Yang, S. Shukla, D. Mathaikutty, and J. Sztipanovits, "Metamodeling: An emerging representation paradigm for system-level design," *Design Test of Computers, IEEE*, vol. 26, no. 3, pp. 54 –69, may-june 2009.

[30] "Definition of model," 26 May 2013. [Online]. Available: http://en.wikipedia.org/wiki/Model

[31] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning," in *Proceedings of the ACM SIGPLAN*, ser. PLDI '03. ACM, 2003, pp. 77–90.

[32] National Instruments, "Labview," 26 May 2013. [Online]. Available: www.ni.com/labview

[33] Mathworks, "Matlab and simulink," 26 May 2013. [Online]. Available: http://www.mathworks.com/

[34] G. Berry, "The constructive semantics of pure esterel." 26 May 2013. [Online]. Available: http://www-sop.inria.fr/esterel.org/

[35] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: a declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 178–188.

[36] IBM, "Ibm rational rhapsody," 26 May 2013. [Online]. Available: http://www.ibm.com/software/awdtools/rhapsody/

[37] D. D. Gajski, J. Peng, A. Gerstlauer, H. Yu, and D. Shin, "System design methodology and tools," *CECS, UC Irvine, Technical Report CECS-TR-03-02*, January 2003.

[38] S. Borkar, "Thousand core chips: a technology perspective," pp. 746–749, 2007.

[39] F. Guderian, *Developing a Design Flow for Embedded Systems*. Jörg Vogt Verlag, 2013.

[40] T. Kogel, A. Haverinen, and J. Altis, "Ocp tlm for architectural modelling," *OCP-IP white-paper*, 2005.

[41] M. Gries, "Methods for evaluating and covering the design space during early design development," *Journal Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131–183, Dec. 2004.

[42] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, 2000, pp. 250–256.

[43] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindquist, "Network on a chip: An architecture for billion transistor era," in *Proceedings of NorChip*, 2000.

[44] G. Sander, "Vcg visualization of compiler graphs," 26 May 2013. [Online]. Available: http://rw4.cs.uni-sb.de/ sander/html/gsvcg1.html

[45] V. Paxson, "Fast lexical analyzer generator, lawrence berkeley laboratory," 26 May 2013. [Online]. Available: http://prdownloads.sourceforge.net/flex/flex-2.5.35.tar.gz

[46] "Bison - gnu parser generator," 26 May 2013. [Online]. Available: http://www.gnu.org/software/bison/

[47] F. Guderian, R. Schaffer, and G. Fettweis, "Administration- and communication-aware ip core mapping in scalable multiprocessor system-on-chips via evolutionary computing," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, june 2012, pp. 1–8.

[48] F. Guderian, R. Schaffer, and G. Fettweis, "Dimensioning the heterogeneous multicluster architecture via parallelism analysis and evolutionary computing," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, june 2012, pp. 1–8.

[49] F. Guderian, E. Fischer, M. Winter, and G. Fettweis, "Fair rate packet arbitration in network-on-chip," in *Proceedings of SOC Conference (SOCC)*, sept. 2011, pp. 278 – 283.

[50] R. Dick, "Embedded system synthesis benchmarks suite," 26 May 2013. [Online]. Available: http://ziyang.eecs.umich.edu/~dickrp/e3s/

[51] EEMBC, "The embedded microprocessor benchmark consortium," 26 May 2013. [Online]. Available: http://www.eembc.org/

```
/**** INCLUDE PREDEFINED FLOWS AND STEPS ****/
#include "parameter_tuning.dfl"
#include "multicluster_load_balancing.dfl"
/**** SEQUENCE IDENTIFIER DEFINITION ****/
int S_TUN = 1; int S_BAL = 2;
/**** DFFINE THE SEQUENCE OF FLOWS ****/
vector<int> flow_order;
flow_order.push_back(S_TUN);
flow_order.push_back(S_BAL);
/**** RUN CONFIGURED FLOWS ****/
for (int i=0; i<flow_order.size(); ++i) {
  string description;
  Flow eslDesignFlow;
  /**** DEFINITION OF FLOW SPECIFIC PARAMETERS
       ****/
  vector<string> arch_in, config;
  /**** SELECT FLOW CONFIGURATION ****/
  switch (flow_order.at(i)) {
    case S_TUN:
    {
      description = "Parameter Tuning";
      arch_in.push_back("lambda_tun/archs/*.
          xml");
      eslDesignFlow = tun;
      break;
    }
    case S_BAL:
    {
      description = "Multicluster Load
          Balancing";
      arch_in = getFilenames("lambda_bal/archs
          /*.xml");
      config = getFilenames("lambda_bal/
          configs/*.xml");
      eslDesignFlow = bal;
      break;
    }
    default:
    {
      println("Unknown Flow Choice: " +
          flow_order.at(i));
      continue;
    }
  }
  /**** EXECUTE SELECTED FLOW ****/
  println(description + " is running ...");
  execute(eslDesignFlow);
}
```

Listing 3. DFL source code for the flow sequence in the case study prototype.

```
/****** ALLOCATION STEP *****/
Step alloc = Step("Allocation");
vector<string> views;
views.push_back("Computation Resources");
views.push_back("Data Logistic Resources");
views.push_back("Resource Management");
alloc.add("View", views);
alloc.add("Execution", "Allocation");
alloc.add("-tool", "IPCoreMapping");
alloc.add("IPCoreMapping", "true");
string alloc_config_param = "-app_in
    lambda_tun\\apps_state_mod.xml -config
    lambda_tun\\dfConfigNoC.xml -
    arch_inlambda_tun\\arch_gen.xml -
    arch_dir_out lambda_tun\\archs -
    mappings_in lambda_tun\\mappings_ideal.xml
    ";
string alloc_static_param = " -AffinityWeight
    0.5 -star_size 2 -rows 3 -columns 3 -r 1 -
    s 50";
alloc.add("Argument", alloc_config_param +
    alloc_static_param);
/***** INPUT PARAMETER SPACE *****/
vector<int> ngen = [1000:10000:1000];
vector<int> popsize = [50:200:50];
vector<int> pmut = [0.01:0.1:0.01];
vector<int> pcross = [0.2:0.4:0.2];
vector<string> space;
space.push_back("ngen");
space.push_back("popsize");
space.push_back("pmut");
space.push_back("pcross");
alloc.add("Space", "space");
alloc.add("Strategy", "ES");
/***** PARALLEL EXECUTION *****/
string parallel = "true";
alloc.add("HPCJob", parallel);
alloc.add("workDirectory", "\\\\server\\hpc");
alloc.add("MaxCores", 15);
alloc.add("scheduler", "entmhpc3");
/***** VALIDATION STEP *****/
Step val = Step("Computation_Validation");
val.add("View", "Computation Resources");
val.add("Execution", "Validation");
val.add("-tool", "Evaluation");
val.add("Objective", "min");
val.add("Metric", "GAFitnessScore");
string val_config_param = "-mappings_dir_in
    lambda_tun\\maps -eval_out lambda_tun\\
    eval_mappings.xml";
val.add("Argument", val_config_param);
/***** FLOW CONSTRUCTION *****/
Flow tun = Flow("Parameter Tuning");
tun.add(alloc);
tun.add(val);
connect(alloc, val);
/***** FLOW VISUALIZATION *****/
tun.save("vcg", "parameter_tuning.vcg");
```

Listing 4. DFL source code for the *Parameter Tuning* flow.